Scala Step-by-Step

Soundness for DOT with Step-Indexed Logical Relations in Iris

PAOLO G. GIARRUSSO, Delft University of Technology, The Netherlands
LÉO STEFANESCO, IRIF, Université de Paris & CNRS, France
AMIN TIMANY, Aarhus University, Denmark
LARS BIRKEDAL, Aarhus University, Denmark
ROBBERT KREBBERS, Delft University of Technology, The Netherlands

The metatheory of Scala's core type system — the *Dependent Object Types (DOT)* calculus — is hard to extend, like the metatheory of other type systems combining subtyping and dependent types. Soundness of important Scala features therefore remains an open problem in theory and in practice. To address some of these problems, we use a *semantics-first* approach to develop a logical relations model for a new version of DOT, called **guarded DOT (gDOT)**. Our logical relations model makes use of an abstract form of *step-indexing*, as supported by the Iris framework, to model various forms of recursion in gDOT. To demonstrate the expressiveness of gDOT, we show that it handles Scala examples that could not be handled by previous versions of DOT, and prove using our logical relations model that gDOT provides the desired data abstraction. The gDOT type system, its semantic model, its soundness proofs, and all examples in the paper have been mechanized in Coq.

CCS Concepts: • **Theory of computation** → **Programming logic**; **Type theory**; **Logic and verification**; **Program verification**.

Additional Key Words and Phrases: DOT, Scala, type soundness, data abstraction, step-indexing, logical relations, Iris, Coq

ACM Reference Format:
Paolo G. Giarrusso, Léo Stefanesco, Amin Timany, Lars Birkedal, and Robbert Krebbers. 2020. Scala Step-by-Step: Soundness for DOT with Step-Indexed Logical Relations in Iris. *Proc. ACM Program. Lang.* 4, ICFP, Article 114 (August 2020), 30 pages. https://doi.org/10.1145/3408996

1 INTRODUCTION

The Scala language has an expressive type system that supports, among other features, first-class recursive modules, path dependent types, impredicative type members, and subtyping, achieving strong information hiding. Alas, Scala has struggled for years with type soundness issues and ad-hoc fixes. To address these issues more rigorously, the compiler of the new Scala 3 language (called Dotty) has been designed hand in hand with a new foundational type system — the *Dependent Object Types (DOT)* calculus. This development led to a number of increasingly expressive versions of DOT and type soundness proofs thereof [Amin et al. 2016; Kabir and Lhoták 2018; Rapoport et al. 2017; Rapoport and Lhoták 2016; Rompf and Amin 2016], culminating in the pDOT calculus [Rapoport and Lhoták 2019], and has helped to fix various soundness bugs in Scala 3 [Rompf and Amin 2016].

Authors' addresses: Paolo G. Giarrusso, Delft University of Technology, The Netherlands; Léo Stefanesco, IRIF, Université de Paris & CNRS, France; Amin Timany, Aarhus University, Denmark; Lars Birkedal, Aarhus University, Denmark; Robbert Krebbers, Delft University of Technology, The Netherlands.

2475-1421/2020/8-ART114
https://doi.org/10.1145/3408996

Despite this exciting development, current DOT versions still lack features necessary to encode full Scala, such as subtyping for recursive types (supported by Amin and Rompf [2017]; Rompf and Amin [2016] but dropped in later work), distributive subtyping [Giarrusso 2019], higher-kinded types [Odersky 2016; Odersky et al. 2016; Stucki 2016, 2017], and mutually recursive modules that hide information from each other (which we dub *mutual information hiding*, and motivate in Sec. 1.1). Worse, one of the core DOT features is support for abstract types and data abstraction, but traditional *syntactic type soundness proofs* cannot show that abstract types behave correctly. Supporting these features in pDOT poses the following questions:

(1) How to design a type system that soundly extends pDOT with these features?
(2) How to prove type soundness of such a type system?
(3) How to demonstrate proper support for data abstraction?

Question (1) is challenging because feature interaction in Scala 3 and (p)DOT is prone to unexpected type soundness issues. Question (2) is challenging because current syntactic type soundness proofs for DOT are intricate, and thus hard to scale to new DOT variants. While Rapoport et al. [2017] describe a recipe for syntactic proofs for DOT, applying this recipe to pDOT — one of the most expressive versions of DOT to date — involves 7 carefully designed variants of pDOT's typing judgment [Rapoport and Lhoták 2019]. Moreover, subtyping for recursive types is not supported by pDOT but, to date, only by the soundness proofs by Rompf and Amin [2016] and Amin and Rompf [2017], which have not been extended in the published literature, and lack other crucial pDOT features. Generally, syntactic type soundness proofs are known to be hard to scale to combinations of (path) dependent types and subtyping [Hutchins 2010; Yang and d. S. Oliveira 2017]. Finally, question (3) is challenging because it cannot be addressed through syntactic type soundness proofs.

To extend pDOT despite these challenges, we eschew traditional syntactic type soundness proofs, and follow a *semantics-first* approach — first, we model each type and typing judgment semantically via a logical relation, *i.e.,* in terms of the program's runtime behavior, instead of a fixed set of syntactic rules. Such a semantic model immediately addresses question (2): only safe programs are semantically typed. To address question (1), we then derive from the semantic model a sound type system called **guarded DOT (gDOT)**: we give modular soundness proofs of rules that either exist in some DOT variant or are suggested by the model, such as those for mutual information hiding (see Sec. 1.1 and Sec. 3 and 4). Some rules, such as subtyping for recursive types, become easier to prove sound than in past work [Amin and Rompf 2017; Rompf and Amin 2016]. Other rules require extending gDOT with a "later" type operator (▷) [Nakano 2000], which enforces certain *guardedness* restrictions obtained from the semantic model (Sec. 4). While these guardedness restrictions make a straightforward translation from pDOT to gDOT impossible (we conjecture a translation exists, but leave it for future work, see Sec. 9), we show that challenging examples from the DOT literature, as well as new examples, can be typed in gDOT. Finally, as we demonstrate in Sec. 1.2 and 6.3, semantic models like ours support proving data abstraction, answering question (3).

1.1 Example: Mutual Information Hiding

Before explaining how to reason formally about data abstraction (Sec. 1.2), the intuitive idea behind semantic typing (Sec. 1.3), and our contributions (Sec. 1.4), we give a brief introduction to Scala and DOT. As a running example we use a novel Scala feature (mutual information hiding) that is supported by Scala compilers and gDOT, but not by prior DOT calculi.

Scala objects enable encoding a rich module system. Objects can contain not only value members (such as fields and methods), but also *type members*, which enables using objects as modules. These type members are *translucent* [Harper and Lillibridge 1994]. That is, their definition can be either exposed or *abstracted away* from clients, supporting a strong form of information hiding. Moreover,

```
object pcore {
  object types {
    abstract class Type
    class TypeTop extends Type
    class TypeRef(val symb: pcore.symbols.Symbol) extends Type {
      assert(!symb.tpe.isEmpty) }
    val typeFromTypeRefUnsafe = (t: types.TypeRef) =>
      // relies on TypeRef's class invariant; only semantically well-typed.
      t.symb.tpe.asInstanceOf[Some[types.Type]].get
  }
  object symbols {
    class Symbol(val tpe: Option[pcore.types.Type], val id: Int)
    // Encapsulation violation, and type error in Scala (but not pDOT)
    // val fakeTypeRef : types.TypeRef =
    //   new { val symb = new Symbol(None, 0) }
  }
}
```

Fig. 1. A (simplified) fragment of the Scala 3 compiler (Dotty), in Scala syntax, that makes use of mutual information hiding and relies on Scala's support for data abstraction for its soundness.

type members can be abstracted away after creation, through *upcasting*. Objects containing type members are first-class values, avoiding the need for a separate module language. Notably, they can be nested, thus supporting *hierarchical* modules, and they can be *mutually recursive*, thus enabling mutually recursive modules. This combination of features enables in particular *mutual information hiding*, that is, mutually recursive modules that hide information from each other.

To demonstrate usefulness of mutual information hiding, consider the example in Fig. 1, adapted from Rapoport and Lhoták [2019], and inspired by the actual implementation of the Scala 3 compiler (Dotty). The example models a system with mutually recursive modules `types` and `symbols`, encoded as members of the object `pcore` and representing separate compilation units. The module `types` represents the API for types. It uses nested classes to model an algebraic data type `Type` for types of the object language,[1] which for simplicity can be either the top-type `TypeTop`, or a reference `TypeRef` to a symbol `symb`. The module `symbols` represents the API for a symbol table, and defines a nested class `Symbol` for symbols, which contain an (optional) type `tpe` and an identifier `id`. Optional types are encoded through the standard type constructor `Option`, with constructors `Some` and `None`, and methods `isEmpty` and `get`. We elaborate on the encoding of `Option` in DOT in Sec. 6.3.

The classes `TypeRef` and `Symbol` have value members (`symb` for `TypeRef`, and `tpe` and `id` for `Symbol`) that are initialized by a corresponding constructor. For instance, after executing `val s = new Symbol(None, 0)`, field `s.id` has value `0`. To achieve strong information hiding, Scala classes are *nominal*, *i.e.,* they can only be constructed through constructors. For instance, Scala rejects `fakeTypeRef`, which creates an object of type `TypeRef` with all the right members (namely, a member `symb` of the right type), because it sidesteps `TypeRef`'s constructor.

Nominality helps to enforce *class invariants* — constructors can validate parameters and initialize objects correctly. For instance, `TypeRef`'s constructor ensures (using `isEmpty`) the invariant that `symb` contains a type. The `fakeTypeRef` method would violate this invariant, but is rejected because it sidesteps `TypeRef`'s constructor. Class invariants can be relied upon by clients. Thanks to the invariants of `Option` and `TypeRef`, clients can assume that `symb.tpe` is never `None`. Indeed, the *unsafe* cast `tpe.asInstanceOf[Some[types.Type]]` in `typeFromTypeRefUnsafe` relies on `TypeRef`'s

[1] For our purposes, an `abstract class` is simply a `class` without constructors.

```
let options = . . .  in let pcore = ν pcore. {
  types = ν types. {
    Type         >: ⊥ = ⊤
    TypeTop      >: ⊥ = types.Type
    newTypeTop   : ⊤ → types.TypeTop = λ_.  ν_. {}
    TypeRef      >: ⊥ = types.Type ∧ {symb : pcore.symbols.Symbol}
    newTypeRef   : pcore.symbols.Symbol → types.TypeRef
                 = λs. { ν_. {symb = s} }
  }
  symbols = ν symbols. {
    Symbol       >: ⊥ = { tpe : options.Option ∧ {A >: ⊥ <: pcore.types.Type};  id : Nat }
    newSymbol    : (options.Option ∧ {A >: ⊥ <: pcore.types.Type}) → Nat → symbols.Symbol
                 = λt i.  ν_. {tpe = t; id = i}
  }
} in . . .
```

Fig. 2. The (simplified) fragment of Dotty from Fig. 1, in pDOT syntax, minus `typeFromTypeRefUnsafe` and the assertion. This code is not well-typed as-is in pDOT (see text).

class invariant to safely extract the **`Type`** nested inside `symb`. While such unsafe casts are not well-typed, they are often used by programmers, who justify their safety by relying on the type system's support for data abstraction. Notably, programmers rely on the type system to reject methods that break nominality (such as `fakeTypeRef`) so that class invariants (such as `!symb.tpe.isEmpty`) are maintained. Moreover, even syntactically well-typed code often relies on class invariants for functional correctness, as encouraged by standard object-oriented practice.

Although Scala can enforce the desired data abstraction in the example, and thus enables programmers to reason informally about their code via class invariants (*e.g.,* to justify the use of unsafe casts), pDOT cannot enforce that. To explain why, we show in Fig. 2 the translation of the example (minus `typeFromTypeRefUnsafe` and the `assert` in **`TypeRef`**'s constructor, to which we come back in Sec. 1.2) into pDOT syntax. As the translation is verbose, we focus on the key aspects.

First, we create objects through syntax $\nu x.\ \{\overline{d}\}$, where x is the *self variable* that refers to the object being created, and $\overline{d}$ is a list of type and value member definitions. The definition of the top-level object `pcore` uses the self variable *pcore* to create the mutual dependency between the subobjects `types` and `symbols`, represented as value members. For brevity, we write the *type declarations* of each member together with their definitions. In the core pDOT syntax, declarations would not appear in object bodies, but in their types — we would write $\nu x.\ \{\overline{d}\} : \mu x.\ \{\overline{T}\}$, where $\{\overline{T}\}$ contains type declarations for all members in $\overline{d}$, which can refer to each other through self variable x.

Second, while (p)DOT does not have native support for higher-kinded types, **`Option`**`[`T`]` can be encoded as $\mathit{options}.\mathsf{Option} \wedge \{\mathsf{A} >: \bot <: T\}$, exposing the type T of elements as type member A.[2]

Third, while classes are native constructs in Scala, they are encoded through abstract types in (p)DOT. To model that classes are nominal (*i.e.,* that they can only be created through constructors), only an upper bound on the abstract type is exposed. Hence, nominality and enforcement of class invariants translate to proper data abstraction. As shorthand, we write $\mathsf{A} >: L = U$ for a type member that is *defined* to be equal to U, but *declared* to have lower bound L and upper bound U. For example, the bounds on TypeRef are $>: \bot <: \mathit{pcore}.\mathtt{types}.\mathsf{Type} \wedge \{\mathtt{symb} : \mathit{pcore}.\mathtt{symbols}.\mathsf{Symbol}\}$.

[2] This encoding of higher-kinded types is insufficient for full Scala [Odersky et al. 2016], motivating the search for higher-kinded DOT [Odersky 2016; Stucki 2016, 2017].

Due to the lower bound ($\bot$, the empty type), clients of `types` cannot construct a TypeRef themselves. The upper bound ($\mathit{pcore}.\texttt{types}.\mathrm{Type} \land \{\texttt{symb} : \mathit{pcore}.\texttt{symbols}.\mathrm{Symbol}\}$) exposes that TypeRef is a subtype of Type, and that it has a value member `symb`.

The code in Fig. 2 properly models the desired information hiding between the recursively defined subobjects `types` and `symbols`. Alas, pDOT cannot type this code, as pDOT requires that all (recursive) objects $\nu(x : T).\,\{\overline{d}\}$ must have a *precise self type* T [Rapoport and Lhoták 2019]. Informally, T is precise if the bounds $>: L <: U$ of all type members that appear hereditarily in T satisfy $L = U$ — *i.e.,* if the recursively defined object does not contain any abstract type members (we define this notion formally in Fig. 4). In this case, the restriction implies that the object *pcore* cannot be typed, for instance because type member TypeRef is imprecise.[3]

The restrictions of pDOT to precise self types appear necessary: pDOT with imprecise self types has known counterexamples to type soundness (see Sec. 3). To the best of our knowledge, the gDOT system, as presented in this paper, is the first DOT variant that supports sound imprecise self types, and thereby provides the desired data abstraction that Scala is supposed to ensure.

1.2 Formal Reasoning About Data Abstraction

As demonstrated in Sec. 1.1, programmers sometimes use the programming language's support for data abstraction to justify the safety of escape hatches such as unsafe casts. We exemplified that in the method `typeFromTypeRefUnsafe` (Fig. 1), which contains an unsafe cast whose safety depends on a class invariant expressing that field `symb.tpe` is never **`None`**. Inspired by the RustBelt project on proving safety of Rust libraries that make use of unsafe code blocks [Jung et al. 2018a, 2020], we use our semantic model to make the informal reasoning from Sec. 1.1 formal. Concretely, we show that code that cannot be syntactically typed because it uses escape hatches whose safety depends on class invariants, can be semantically typed using a manual proof in gDOT's semantic model. The ability to carry out such manual proofs demonstrates gDOT's support for data abstraction.[4]

1.3 The Semantics-First Approach

To formally investigate challenging Scala features (such as imprecise self types), to develop a modular approach to prove type soundness, and to study Scala's support for data abstraction, we approach the problem of designing a suitable DOT calculus *semantics-first*. That is, we *first* design a semantic model, and *then* derive from it a sound type system, namely gDOT.

Our model is based on an old idea going back to at least Milner [1978]: we formalize the meaning of DOT types *semantically* (using logical relations) by mapping syntactic types T to *semantic types* $\mathcal{V}[\![\, T \,]\!] \in \mathrm{SemType}$. Semantic types $\mathrm{SemType} \triangleq (\mathrm{Var} \rightarrow \mathrm{SemVal}) \rightarrow \mathrm{SemVal} \rightarrow \mathrm{Prop}$ are predicates on values (or equivalently, functions from values to propositions) that take as a parameter an environment mapping variables to values (since types can contain variables that point to values). We then show that if a closed term has a certain type T, then any result of evaluating that term satisfies $\mathcal{V}[\![\, T \,]\!]_\varnothing$ (the *fundamental theorem*). However, we need a novel idea to handle abstract types, in particular because DOT type members are *impredicative*: that is, type members can describe values containing in turn type members, without any stratification (see Sec. 8).

To explain our gDOT model, we first sketch a naive semantics that is simple, but *unsound* because of DOT impredicativity. We then explain how we can use *step-indexing* [Appel and McAllester 2001], a common technique to deal with circularities, to give a more refined but sound model.

[3] In pDOT one can construct the top-level object `pcore` with a precise self type, and only after it is constructed use subsumption to weaken the bounds on the type members. This way, one can achieve information hiding for clients of `pcore`, but not information hiding between `types` and `symbols`. Hence, this is not a complete solution.

[4] Beware we do *not* propose extending the Scala type checker to accept such programs. Instead, we propose a formal foundation for the informal reasoning often used to review uses of escape hatches.

In (g)DOT, ignoring both base values and paths, a value can be a variable, a function value (a λ-abstraction), or an *object* (a finite map from member labels to semantic types or values). If we think of semantic types as predicates (*i.e.,* functions from values to propositions Prop), then we can describe such values using a recursive domain equation of the following form:

$$\begin{aligned} \mathsf{SemType} &\triangleq (\mathsf{Var} \to \mathsf{SemVal}) \to \mathsf{SemVal} \to \mathsf{Prop} \\ \mathsf{SemVal} &\cong \mathsf{Var} + \left\{ \lambda x.\, e \right\} + (\mathsf{Label} \xrightarrow{\text{fin}} (\mathsf{SemType} + \mathsf{SemVal})) \end{aligned} \tag{Domain-Bad}$$

Intuitively, such a naive semantics would justify (p)DOT. But it would also be *unsound*, because it is well-known that there are no solutions to the above recursive domain equation in ordinary set theory due to the negative recursive occurrences of SemVal (visible after unfolding SemType).

Luckily, we can use the Iris logic [Jung et al. 2016, 2018b, 2015; Krebbers et al. 2017a] and an abstract form of step-indexing [Appel et al. 2007; Birkedal et al. 2011] to stratify our definition and build a sound version of this semantics. Our stratified equation is written as follows:

$$\begin{aligned} \mathsf{SemType} &\triangleq (\mathsf{Var} \to \mathsf{SemVal}) \to \mathsf{SemVal} \to \mathsf{iProp} \\ \mathsf{SemVal} &\cong \mathsf{Var} + \left\{ \lambda x.\, e \right\} + (\mathsf{Label} \xrightarrow{\text{fin}} (\blacktriangleright\mathsf{SemType} + \mathsf{SemVal})) \end{aligned} \tag{Domain}$$

Here, iProp is the universe of Iris propositions. Now the negative occurrences of SemVal are *guarded* by a "later" ►, a contractive type operator that restricts how its argument can be manipulated. Formally, this can be understood and solved as a recursive domain equation in the category of complete ordered families of equivalences (COFEs) [America and Rutten 1989; Birkedal et al. 2010].

Using the solution to the recursive domain equation we obtain a sound model for gDOT. We consider this model "canonical", as we have taken a straightforward but naive semantics, and done the smallest possible change to turn it into a sound semantics using step-indexing. Interestingly, our model differs in a number of key points from prior work on step-indexed logical relations:

- Prior work focused mostly on modeling general references, and thus had to solve the so-called "type-world circularity" [Ahmed 2004; Birkedal et al. 2011]. Since such type systems in prior work do not support dependent types, values cannot contain types, and as such, the domain of values was not recursively defined (it was simply the set of syntactic values).
- While Iris could in principle solve the aforementioned recursive domain equation, we actually represent semantic values through an indirection, which we call *stamping*. This keeps the syntax of values first-order (which aids mechanization in Coq), and enables reuse of Iris's support for *saved predicates* [Jung et al. 2016].
- Our logical relation combines non-termination (for terms) and termination (for paths).

Our model also differs from Wang and Rompf's logical relations model [2017] for normalization of a DOT subset (also discussed in Sec. 8). Their DOT subset excludes value members (so cannot express the example from Fig. 1). While they also model impredicative type members, they define an explicitly-stratified logical relation, instead of using Iris's abstract form of step-indexing, which is mathematically better behaved and enjoys all the Iris infrastructure.

1.4 Contributions

To sum up, we take a *semantics-first* approach to take a fresh look at several open problems of Scala's core calculus pDOT. Through the semantics-first approach we obtain the new **guarded DOT (gDOT)** calculus, which enforces certain *guardedness* restrictions by extending the type system with a "later" operator (▷). This operator makes it possible to add a number of novel and provably sound typing rules, *e.g.,* to support imprecise self types and mutual information hiding, that were unsound in prior versions of DOT. Unfortunately, gDOT's guardedness restrictions come with a price — many programs that were accepted by previous (p)DOT versions require additional uses of later. We conjecture all pDOT programs can be translated, but leave a proof for future work. Instead, we demonstrate that we can encode many challenging examples from the Scala and (p)DOT literature, as well as new examples that could not be handled before.

Concretely, this paper makes the following contributions:

- We motivate extending pDOT with support for *imprecise self types*, to enable type abstractions between mutually recursive objects, despite the known difficulties (Sec. 1.1 and Sec. 3).
- After summarizing the pDOT calculus (Sec. 2), we introduce our new gDOT calculus (Sec. 4).
- We present a novel technique, based on step-indexed logical relations in Iris, to give a semantic model of impredicative type members, and use it to prove soundness of gDOT (Sec. 5).
- We demonstrate gDOT's expressivity by encoding various examples, and demonstrate its support for data abstraction by proving semantic typing of functions whose correctness relies on gDOT's ability to maintain class invariants (Sec. 6).
- We mechanize gDOT and all proofs in this paper in Coq using the Iris framework (Sec. 7). The Coq mechanization can be found online [Giarrusso et al. 2020].

2 BACKGROUND: PDOT

Before we present gDOT in Sec. 4, we summarize pDOT [Rapoport and Lhoták 2019] — our starting point. To simplify the comparison, we reformulate pDOT to be closer to gDOT, but preserving its essence. Like all DOT calculi, pDOT and gDOT are neither intended for programming directly, nor designed for decidable type checking, but rather as an elaboration target for type-preserving translation from subsets of Scala.

2.1 Syntax and Operational Semantics

pDOT and gDOT share syntax and operational semantics, which are shown in Fig. 3 (ignoring primitives like numerals and addition). Unlike in Sec. 1.3, we define syntactic values and types, not semantic ones (we return to semantic types and values in Sec. 5.3). pDOT values are either variables x, functions $\lambda x.\, e$ or objects $\nu x.\, \{\overline{d}\}$ (where ν is distinct from v). An object contains a map from labels to definitions $\overline{d}$, which can reference the whole object through the *self variable* x, modeling the `this` variable in Scala. A definition d can be a *type member* $\{\mathsf{A} = T\}$, where A is a type label and T is a type, or a *term member* $\{\mathsf{a} = p\}$, where a is a label and p is a *(pre)path*. Though they are central to the type system, type members do not affect the operational semantics. Type and term members can be projected from objects using *member selectors*, respectively $e.\mathsf{a}$ and $p.\mathsf{A}$. A path is either a value v or a selection $p.\mathsf{a}$. Nonsensical paths such as $(\lambda x.\, e).\mathsf{a}$ are rejected by typing.

Like in *storeless DOT* [Amin 2016, Ch. 3], we use a conventional substitution-based call-by-value semantics. Substitution of variables by *values* is written as $\chi[x := v]$, where χ ranges over all syntactic classes. We write $e \to_{\mathsf{h}} e'$ for head reduction, and write $e \to_{\mathsf{t}} e'$ for its closure under call-by-value evaluation contexts K. Head reduction has three rules: the usual call-by-value β-reduction for function values, evaluation of member selectors, and evaluation of coercions **coerce**. As DOT objects are recursive, the member lookup relation $v.l \searrow d$ for an object $v = \nu x.\, \{\overline{d}\}$ substitutes the self variable x by v before looking up l in the substitution result. Last, coercions applied to values simply reduce away in one evaluation step, which will become significant in Sec. 5.3. Coercions get their name because they appear in gDOT's subsumption rule (T-Sub) in Fig. 6.

2.2 Type System

We now present the pDOT (pre)types (Fig. 3) and type system (Fig. 4 and 5). We focus on the typing rules that are essential to the rest of the paper, and defer to Rapoport and Lhoták [2019] for the remaining ones. A term in pDOT can be typed either with a dependent function type $\forall x : S.\, T$,

Syntax

$$
\begin{array}{rll}
\mathsf{TyLabel} \ni \mathsf{A} & & \textit{Type member labels} \\
\mathsf{ValLabel} \ni \mathsf{a} & & \textit{Term member labels} \\
\mathsf{Label} \ni l & ::= \mathsf{a} \mid \mathsf{A} & \textit{Member labels} \\
\mathsf{Val} \ni v & ::= x \mid \lambda x.\, e \mid \nu x.\, \{\bar{d}\} & \textit{Values} \\
\mathsf{Term} \ni e & ::= v \mid e\ e \mid e.\mathsf{a} \mid \textbf{coerce}\ e & \textit{Terms} \\
\mathsf{Path} \ni p, q & ::= v \mid p.\mathsf{a} & \textit{(Pre)paths} \\
\mathsf{DefBody} \ni d & ::= p \mid T & \textit{Definition bodies} \\
\mathsf{DefList} \ni \bar{d} & ::= l = d \mid \bar{d}; \bar{d} & \textit{Definition lists} \\
\mathsf{ECtx} \ni K & ::= [] \mid K\ e \mid v\ K \mid K.\mathsf{a} \mid \textbf{coerce}\ K & \textit{Evaluation contexts} \\
\mathsf{Type} \ni L, S, T, U, V, W & ::= \top \mid \bot \mid T \wedge U \mid T \vee U \mid \forall x : S.\, T & \textit{(Pre)types} \\
& \quad \mid \{\mathsf{a} : T\} \mid \{\mathsf{A} >: L <: U\} \mid p.\mathsf{A} \mid p.\textbf{type} \mid \mu x.\, T \mid \triangleright T & \\
\mathsf{TyCtx} \ni \Gamma & ::= \varepsilon \mid \Gamma, x : T & \textit{Typing contexts}
\end{array}
$$

Member selection (looking up label l in value v finds definition d)

$$v.l \searrow d \triangleq \exists x, \bar{d}.\ v = \nu x.\, \{\bar{d}\} \wedge \mathsf{lookup}\ l\ (\bar{d}[x := v]) = d$$

Operational semantics (call-by-value head reduction $e \to_{\mathsf{h}} e'$, and its closure $e \to_{\mathsf{t}} e'$ over contexts)

$$(\lambda x.\, e)\ v \to_{\mathsf{h}} e[x := v] \qquad \frac{v.a \searrow p}{v.\mathsf{a} \to_{\mathsf{h}} p} \qquad \textbf{coerce}\ v \to_{\mathsf{h}} v \qquad \frac{e \to_{\mathsf{h}} e'}{K[e] \to_{\mathsf{t}} K[e']}$$

Fig. 3. pDOT/gDOT syntax and operational semantics. New gDOT constructs have a shaded background. The original pDOT path syntax replaces values by variables.

where x can appear in T, a record type $\{\mathsf{a} : T\}$ or $\{\mathsf{A} >: L <: U\}$, a path selection $p.\mathsf{A}$, a singleton type $p.\textbf{type}$, or an object type $\mu x.\, T$. In addition, pDOT features the usual top ($\top$), bottom ($\bot$), and intersection ($\wedge$) types.[5] We use $\{\bar{T}\}$ as sugar for intersections, *e.g.,* we let $\{\mathsf{A} >: L <: U;\ \mathsf{a} : T\}$ be syntactic sugar for $\{\mathsf{A} >: L <: U\} \wedge \{\mathsf{a} : T\}$. Distinct members $\bar{T}$ of type $\mu x.\, \{\bar{T}\}$ cannot refer to each other directly, but only through the self variable x.

The typing judgments contain a context Γ, which is a list of mappings $x : T$ from variables x to types T. Contexts are dependently typed, but with non-standard scoping: in context $\Gamma_1, x : T, \Gamma_2$, the variable x is bound not only in Γ_2, but also in T. Beyond correct scoping, DOT calculi do not enforce any well-formedness requirement on either type members (in (D-Typ)), types or contexts.

The term typing judgment $\Gamma \vdash e : T$ (which also covers values), and subtyping judgment $\Gamma \vdash T_1 <: T_2$, as well as the subsumption rule (T-Sub), are standard. We write $\Gamma \vdash T_1 <: T_2 <: T_3$ for having both $\Gamma \vdash T_1 <: T_2$ and $\Gamma \vdash T_2 <: T_3$. In our reference version of DOT, we follow WadlerFest DOT [Amin et al. 2016] and pDOT by leaving out subtyping rules for recursive types, which we will add back in gDOT in Sec. 4. Dependent function types $\forall x : S.\, T$ support standard rules for contravariant subtyping and introduction. Non-dependent functions can be applied to an arbitrary argument term using (T-$\forall$-E). Dependent functions can only be applied to a path p using (T-$\forall$-E$_p$) as we can only substitute paths into types using path substitution $T[x := p]$.

Typing of object values $\nu x.\, \{\bar{d}\}$ in rule (T-{}-I) depends on the *definition typing judgment* $\Gamma \mid x : V \vdash \{\bar{d}\} : T$. Here, the binding for the self variable $x : V$ (which refers to the object being constructed) is placed in a stoup (*i.e.,* one-element context) instead of the context Γ because it has a special role

[5] Some versions of DOT [Rompf and Amin 2016] have union ($\vee$) types, but pDOT does not. We readd union types in gDOT.

Term typing $\boxed{\Gamma \vdash e : T}$

$$\frac{\Gamma \vdash e : T_1 \qquad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash e : T_2}\ (\textsc{T-Sub}) \qquad \frac{\Gamma \vdash_{\mathsf{P}} p : T}{\Gamma \vdash p : T}\ (\textsc{T-Path}) \qquad \frac{\Gamma \mid x : T \vdash \{\bar{d}\} : T}{\Gamma \vdash \nu x.\,\{\bar{d}\} : \mu x.\,T}\ (\text{T-}\{\}\text{-I})$$

$$\frac{\Gamma \vdash e : \{\mathsf{a} : T\}}{\Gamma \vdash e.\mathsf{a} : T}\ (\text{T-}\{\}\text{-E}) \qquad \frac{\Gamma, x : S \vdash e : T \qquad x \notin \mathsf{FV}(S)}{\Gamma \vdash \lambda x.\,e : \forall x : S.\,T}\ (\text{T-}\forall\text{-I})$$

$$\frac{\Gamma \vdash e_1 : S \rightarrow T \qquad \Gamma \vdash e_2 : S}{\Gamma \vdash e_1\ e_2 : T}\ (\text{T-}\forall\text{-E}) \qquad \frac{\Gamma \vdash e : \forall z : S.\,T \qquad \Gamma \vdash_{\mathsf{P}} p : S}{\Gamma \vdash e\ p : T[z := p]}\ (\text{T-}\forall\text{-E}_p)$$

Path typing $\boxed{\Gamma \vdash_{\mathsf{P}} p : T}$

$$\frac{x : T \in \Gamma}{\Gamma \vdash_{\mathsf{P}} x : T}\ (\textsc{P-Var}) \qquad \frac{\Gamma \vdash_{\mathsf{P}} p : T[x := p]}{\Gamma \vdash_{\mathsf{P}} p : \mu x.\,T}\ (\text{P-}\mu\text{-I}) \qquad \frac{\Gamma \vdash_{\mathsf{P}} p : \mu x.\,T}{\Gamma \vdash_{\mathsf{P}} p : T[x := p]}\ (\text{P-}\mu\text{-E}) \qquad \frac{\Gamma \vdash_{\mathsf{P}} p : T_1 \qquad \Gamma \vdash_{\mathsf{P}} p : T_2}{\Gamma \vdash_{\mathsf{P}} p : T_1 \wedge T_2}\ (\text{P-}\wedge\text{-I})$$

$$\frac{\Gamma \vdash_{\mathsf{P}} p : T \qquad \Gamma \vdash T <: U}{\Gamma \vdash_{\mathsf{P}} p : U}\ (\textsc{P-Sub}) \qquad \frac{\Gamma \vdash_{\mathsf{P}} p : \{\mathsf{a} : T\}}{\Gamma \vdash_{\mathsf{P}} p.\mathsf{a} : T}\ (\textsc{P-Fld-E}) \qquad \frac{\Gamma \vdash_{\mathsf{P}} p.\mathsf{a} : T}{\Gamma \vdash_{\mathsf{P}} p : \{\mathsf{a} : T\}}\ (\textsc{P-Fld-I})$$

$$\frac{\Gamma \vdash_{\mathsf{P}} p : q.\mathbf{type} \quad \Gamma \vdash_{\mathsf{P}} q : T}{\Gamma \vdash_{\mathsf{P}} p : T}\ (\textsc{P-Sngl-Trans}) \qquad \frac{\Gamma \vdash_{\mathsf{P}} p : q.\mathbf{type} \quad \Gamma \vdash_{\mathsf{P}} q.\mathsf{a} : T}{\Gamma \vdash_{\mathsf{P}} p.\mathsf{a} : q.\mathsf{a}.\mathbf{type}}\ (\textsc{P-Sngl-E})$$

Definition typing $\boxed{\Gamma \mid x : V \vdash \{\bar{d}\} : T}$

$$\frac{\Gamma, x : V \vdash v : T \qquad \mathbf{tight}\ T}{\Gamma \mid x : V \vdash \{\mathsf{a} = v\} : \{\mathsf{a} : T\}}\ (\textsc{D-Val}) \qquad \frac{\Gamma, x : V \mid z : x.\mathsf{a}.\mathbf{type} \wedge T \vdash \{\bar{d}\} : T \qquad \mathbf{tight}\ T}{\Gamma \mid x : V \vdash \{\mathsf{a} = \nu z.\,\{\bar{d}\}\} : \{\mathsf{a} : \mu z.\,T\}}\ (\textsc{D-Val-New})$$

$$\frac{}{\Gamma \mid x : V \vdash \{\mathsf{A} = T\} : \{\mathsf{A} >: T <: T\}}\ (\textsc{D-Typ}) \qquad \frac{\Gamma, x : V \vdash_{\mathsf{P}} p : T}{\Gamma \mid x : V \vdash \{\mathsf{a} = p\} : \{\mathsf{a} : p.\mathbf{type}\}}\ (\textsc{D-Path-Sngl})$$

$$\frac{\Gamma \mid x : V \vdash \{\bar{d}_1\} : T_1 \qquad \Gamma \mid x : V \vdash \{\bar{d}_2\} : T_2 \qquad \mathrm{dom}\,\bar{d}_1, \mathrm{dom}(\bar{d}_2)\ \text{disjoint}}{\Gamma \mid x : V \vdash \{\bar{d}_1; \bar{d}_2\} : T_1 \wedge T_2}\ (\textsc{D-And})$$

Tight **(or *precise*) types** $\boxed{\mathbf{tight}\ T}$

$$\mathbf{tight}\ T = \begin{cases} L = U & \text{if } T = \{\mathsf{A} >: L <: U\} \\ \mathbf{tight}\ U & \text{if } T = \mu(x : U) \text{ or } T = \{\mathsf{a} : U\} \\ \mathbf{tight}\ U \text{ and } \mathbf{tight}\ V & \text{if } T = U \wedge V \\ \mathsf{True} & \text{otherwise} \end{cases}$$

Fig. 4. pDOT rules for term typing, path typing, and definition typing. Path typing is a special case of term typing in (p)DOT, but not in our presentation or in gDOT.

in rules (D-Val) and (D-Val-New) to type value definitions. These rules require constructed objects to have *precise* self types by using Rapoport and Lhoták's predicate **tight** T [2019].

Paths p are the only terms that can appear in types, through selections $p.\mathsf{A}$ and singleton types $p.\mathbf{type}$. Paths p that appear in types always start with a variable, *i.e.,* they do not contain values. Paths are typed using the *path typing judgment* $\Gamma \vdash_{\mathsf{P}} p : T$, which, unlike term typing, also guarantees that p terminates. Intuitively, a type selection $p.\mathsf{A}$ refers to the type definition for member A in the result of p. However, rules (Sel-<:) and (<:-Sel) relate $p.\mathsf{A}$ only to the upper and

Top, bottom, and intersection types $\boxed{\Gamma \vdash T_1 <: T_2}$

$$\Gamma \vdash T <: \top \;(\text{<:-}\top) \qquad \Gamma \vdash T_1 \wedge T_2 <: T_1 \;(\wedge_1\text{-<:}) \qquad \Gamma \vdash T_1 \wedge T_2 <: T_2 \;(\wedge_2\text{-<:}) \qquad \Gamma \vdash \bot <: T \;(\bot\text{-<:})$$

$$\frac{\Gamma \vdash T <: U_1 \qquad \Gamma \vdash T <: U_2}{\Gamma \vdash T <: U_1 \wedge U_2}\;(\text{<:-}\wedge) \qquad \Gamma \vdash T <: T \;(\text{<:-}\textsc{Refl}) \qquad \frac{\Gamma \vdash S <: T \qquad \Gamma \vdash T <: U}{\Gamma \vdash S <: U}\;(\text{<:-}\textsc{Trans})$$

Type members

$$\frac{\Gamma \vdash_{\mathbf{p}} p : \{\mathsf{A} >: L <: U\}}{\Gamma \vdash L <: p.\mathsf{A}}\;(\text{<:-}\textsc{Sel}) \qquad \frac{\Gamma \vdash_{\mathbf{p}} p : \{\mathsf{A} >: L <: U\}}{\Gamma \vdash p.\mathsf{A} <: U}\;(\textsc{Sel}\text{-<:})$$

Co/contra-variant subtyping

$$\frac{\Gamma \vdash T_1 <: T_2}{\Gamma \vdash \{\mathsf{a} : T_1\} <: \{\mathsf{a} : T_2\}}\;(\textsc{Fld}\text{-<:-}\textsc{Fld}) \qquad \frac{\Gamma \vdash L_2 <: L_1 \qquad \Gamma \vdash U_1 <: U_2}{\Gamma \vdash \{\mathsf{A} >: L_1 <: U_1\} <: \{\mathsf{A} >: L_2 <: U_2\}}\;(\textsc{Typ}\text{-<:-}\textsc{Typ})$$

$$\frac{\Gamma \vdash S_2 <: S_1 \qquad \Gamma, x : T_2 \vdash T_1 <: T_2}{\Gamma \vdash \forall x : S_1.\, T_1 <: \forall x : S_2.\, T_2}\;(\forall\text{-<:-}\forall)$$

Singleton types

$$\frac{\Gamma \vdash_{\mathbf{p}} p : q.\mathbf{type} \qquad T_1 \cong^{*}_{p:=q} T_2}{\Gamma \vdash T_1 <: T_2}\;(\textsc{Sngl}_{pq}\text{-<:}) \qquad \frac{\Gamma \vdash_{\mathbf{p}} p : q.\mathbf{type} \qquad T_1 \cong^{*}_{p:=q} T_2}{\Gamma \vdash T_2 <: T_1}\;(\textsc{Sngl}_{qp}\text{-<:})$$

Fig. 5. pDOT rules for subtyping.

lower bound of A in the type of p, not the definition of A: this ensures that abstract types are indeed abstract. Finally, (Typ-<:-Typ) enables making a type member of p (more) abstract, by upcasting p.

Intuitively, the singleton type $p.\mathbf{type}$ contains a value v if path p is statically guaranteed to evaluate to v. We say that two paths p and q *alias each other* when $\Gamma \vdash_{\mathbf{p}} p : q.\mathbf{type}$ is derivable, which guarantees that both p and q evaluate to the same value. Aliases can be created among others using rule (D-Val-New), which combines (D-Val) and (T-{}-I), but also records that z and $x.\mathsf{a}$ are aliases. Aliasing can be *hidden* through (P-Sngl-Trans): if x is initialized with p, but x's type is not a subtype of $p.\mathbf{type}$, then x will not alias p. The rules ($\textsc{Sngl}_{pq}$-<:) and ($\textsc{Sngl}_{qp}$-<:) use the relation $T \cong^{*}_{p:=q} U$, which is the reflexive transitive closure of Rapoport and Lhoták's path replacement [2019]. Informally, $T \cong^{*}_{p:=q} U$ means that zero or more occurrences of p in T are replaced by q in U, with the rest of T, including any other occurrence of p, left unchanged.

3 UNSOUNDNESS OF DOT WITH IMPRECISE SELF TYPES

As discussed in Sec. 1.1, imprecise self types are useful to support certain forms of information hiding, but current versions of DOT do not support them soundly. In this section we indicate why the restriction to *tight* (or *precise*) types (see Fig. 4) excludes the example in Fig. 2 from Sec. 1.1, but is necessary for soundness of current versions of DOT.

To construct a typing derivation for the example in Fig. 2 we initially give type members such as $\{\mathsf{TypeRef} = (\mathit{pcore}.\mathtt{types}.\mathsf{Type} \wedge \{\mathtt{symb} : \mathit{pcore}.\mathtt{symbols}.\mathsf{Symbol}\})\}$ a concrete type, *i.e.,* we give them exact lower and upper bounds. This is needed to type constructors such as `newTypeRef`, as their bodies need to know the concrete types of the objects they construct. The bodies of `types` and `symbols` are then typed using (T-{}-I). Since these bodies still contain concrete types, we upcast them using subsumption (T-Sub) to types T and S such that type members like TypeRef are abstract, *i.e.,* given the correct lower and upper bound. Finally, we need (D-Val) to type $\{\mathtt{types} : T\}$ and $\{\mathtt{symbols} : S\}$, but this is impossible — since T and S contain abstract types, they violate the **tight** T side-condition on (D-Val), which prevents typing this example.

Unfortunately, as shown by Rapoport and Lhoták [2019], removing this side-condition is unsound, similarly to other desirable generalizations of DOT rules. We discuss two such generalizations:

- Seemingly, to type our example, one could type $\{\mathtt{types} : T'\}$ and $\{\mathtt{symbols} : S'\}$ with tight types S' and T' using (D-Val), and upcast them via subsumption to non-tight types S and T. But this attempt fails, because DOT lacks the following subsumption rule for term members:

$$\frac{\Gamma \mid x : V \vdash \{\mathsf{a} = p\} : \{\mathsf{a} : T_1\} \qquad \Gamma, x : V \vdash T_1 <: T_2}{\Gamma \mid x : V \vdash \{\mathsf{a} = p\} : \{\mathsf{a} : T_2\}}\ (\textsc{D-Path-Sub-Bad})$$

  Amin [2016, Sec. 3.5.5] showed that such a rule is unsound.
- The rule (D-Typ) restricts type members $\{\mathsf{A} = T\}$ to have tight types $\{\mathsf{A} >: T <: T\}$. One may wonder if this rule could be generalized to non-tight bounds as follows:

$$\frac{\Gamma, x : V \vdash L <: T \qquad \Gamma, x : V \vdash T <: U}{\Gamma \mid x : V \vdash \{\mathsf{A} = T\} : \{\mathsf{A} >: L <: U\}}\ (\textsc{D-Typ-Abs-Bad})$$

  Amin [2016, Sec. 3.5.5] showed that such a rule is unsound as well.

All these rules break type soundness similarly. A closed value with type $\{\mathsf{A} >: L <: U\}$ is a *witness* that $L <: U$, and allows upcasting L to U. Closed values with *bad bounds* [Amin 2016], such as $>: \top <: \bot$, enable casting values across arbitrary types, and thus break type soundness. All of the aforementioned rules enable constructing closed values with type $\mu\_.\ \{\mathsf{A} >: \top <: \bot\}$, from which one can deduce the inconsistent subtyping $\top <: \bot$. For example, using unsound rule (D-Typ-Abs-Bad) displayed above, one can show that (for any type S) value $\nu x.\ \{\mathsf{A} = S\}$ has said type:

$$\frac{\dfrac{x : \{\mathsf{A} >: \top <: \bot\} \vdash \top <: S <: \bot}{\varepsilon \mid x : \{\mathsf{A} >: \top <: \bot\} \vdash \{\mathsf{A} = S\} : \{\mathsf{A} >: \top <: \bot\}}\ (\textsc{D-Typ-Abs-Bad})}{\varepsilon \vdash \nu x.\ \{\mathsf{A} = S\} : \mu x.\ \{\mathsf{A} >: \top <: \bot\}}\ (\text{T-\{\}-I})$$

The premise of (T-{}-I) extends the context with an unsound subtyping witness, the self variable x, which enables proving $\top <: \bot$ and (unsoundly) that type definition $\{\mathsf{A} = S\}$ is between its bounds.

To rule out such unsound circular derivations, all previous calculi in the DOT family use the same solution — they restrict object creation to tight (*i.e.,* precise) self types, so that rule (T-{}-I) becomes sound. If T is a precise self type, it can only carry proofs for subtypings of the form $U <: U$, which are always true. To enforce this restriction, DOT puts the **tight** T side-condition on (D-Val), and eschews rules like (D-Path-Sub-Bad) and (D-Typ-Abs-Bad). While this ensures soundness, it rules out imprecise self types, and therefore useful forms of data abstraction.

Our gDOT calculus takes a different route — it imposes a *guardedness* condition on the self variable to ensure it is not used in circular way. Hence, we can soundly support imprecise self types, *i.e.,* allow variants of rules like (D-Val) without the **tight** T side-condition, and (D-Path-Sub-Bad) and (D-Typ-Abs-Bad). To realize such a guardedness condition, we give the self variable a different and *weaker* type. Since DOT provides no suitable candidate, we will extend the language of DOT types. These changes enable us to type examples including the one in Fig. 2 from Sec. 1.1.

Does this make Scala unsound? One might wonder if the aforementioned counterexamples to type soundness affect Scala's support for imprecise self types; but we are unable to encode the counterexamples in Scala. To the best of our understanding, that is because counterexamples, like $\nu x.\ \{\mathsf{A} = S\}$ from this section, rely on transitivity of subtyping to deduce $x : \{\mathsf{A} >: L <: U\} \vdash L <: U$ from $x : \{\mathsf{A} >: L <: U\} \vdash L <: x.\mathsf{A} <: U$. This use of transitivity is not admissible the Scala compiler's (*i.e.*, Dotty's) algorithmic type system [Hu and Lhoták 2020; Nieto 2017]. Nevertheless, it is not at all clear that all such counterexamples are forbidden by Dotty, nor how to prove soundness of imprecise self types by relying on the absence of transitivity.

4 THE GDOT TYPE SYSTEM

The typing rules and subtyping rules of gDOT are displayed in Fig. 6 and 7, and discussed in this section. To support imprecise self types while avoiding the soundness problems from Sec. 3, our guarded DOT (gDOT) calculus imposes a *guardedness condition* that ensures that the self variable x in recursive objects $\nu(x : T).\,\{\overline{d}\}$ is not used in a cyclic way. We enforce this condition by extending DOT with a "later" type operator ($\triangleright$), so that x can be given type $\triangleright T$ instead of T. The type $\triangleright T$ is weaker than T, in the sense that it cannot be used directly in the construction of the object's body $\overline{d}$. Instead, one needs to take a program step to eliminate the later, *i.e.,* to turn $\triangleright T$ into T. Some of the most prominent places where the later type ($\triangleright$) appears in gDOT are:

$$\frac{\Gamma \mid x : \triangleright T \vdash \{\overline{d}\} : T}{\Gamma \vdash \nu x.\,\{\overline{d}\} : \mu x.\,T} \qquad \frac{\Gamma, x : V, z : S \vdash t : T \qquad z \notin \mathsf{FV}(S)}{\Gamma \mid x : \triangleright V \vdash \{\mathsf{a} = \lambda z.\,t\} : \{\mathsf{a} : \forall z : S.\,T\}} \qquad \frac{\Gamma \vdash e : \triangleright T}{\Gamma \vdash \textbf{coerce}\ e : T}$$

The first rule is gDOT version of the introduction rule for recursive objects (T-{}-I), which puts $x : \triangleright V$ instead of $x : V$ in the context. The later can be eliminated using the other rules — either by constructing a function, or by taking an explicit step using gDOT's **coerce** construct.

Our later type operator ($\triangleright$) *reflects* into gDOT Iris's later modality ($\triangleright$) (see Sec. 5.3), which can be eliminated by taking a program step. We also rely on the principle of *Löb induction*, which allows proving proposition P under induction hypothesis $\triangleright P$, and enables proving (T-{}-I) sound.

By ensuring that self variables are only used in a guarded way, we can remove the **tight** T side-condition of rules (D-Val) and (D-Val-New), generalize (D-Path-Sngl) to (D-Path), and add sound versions of (D-Typ-Abs-Bad) and (D-Path-Sub-Bad) to gDOT. Such rules become sound because rule (T-{}-I) only types the self variable as $\triangleright V$, and thereby prevents unsound circular derivations. Notably, the counterexamples from Sec. 3 are ruled out because it is impossible to deduce $\top <: \bot$ from $x : \triangleright \{\mathsf{A} >: \top <: \bot\}$, just like one cannot deduce False from $\triangleright \mathsf{False}$ in step-indexed logics.

gDOT's (D-Val-New) enables typing the example in Fig. 2 without changes: imprecise self types enable mutual information hiding between the definitions of TypeRef and Symbol.

This section continues as follows. To deal with laters in types of paths, we generalize the subtyping and path typing judgments to their *delayed* variants (Sec. 4.1). We explain the guardedness restrictions gDOT imposes on type selectors (Sec. 4.2), and how laters can be eliminated through function introduction (Sec. 4.3). We show that in addition to aforementioned new rules, gDOT also supports a number of other rules that prior versions of DOT did not support (Sec. 4.4).

4.1 Delayed Path Judgments

Laters can be eliminated from terms using rule (T-Coerce), which says that $\Gamma \vdash e : \triangleright T$ implies $\Gamma \vdash \textbf{coerce}\ e : T$. Since paths are required to terminate, they do not have a corresponding **coerce** construct — as we will explain in Sec. 5.2.3, the concept of eliminating a later is incompatible with termination. Instead, we extend path typing $\Gamma \vdash^{i}_{\mathsf{p}} p : T$ with a so called *delay* $i \in \mathbb{N}$ that allows to accumulate laters. This idea becomes evident by rule (P-Later), which says that $\Gamma \vdash^{i}_{\mathsf{p}} p : \triangleright T$ implies $\Gamma \vdash^{i+1}_{\mathsf{p}} p : T$. In Sec. 4.2 we see how laters arise in paths, and why this rule is useful.

Similar to path typing, we also equip subtyping $\Gamma \vdash^{i} S <: T$ with a delay $i \in \mathbb{N}$. Most of the delayed subtyping rules are generalizations of pDOT rules to arbitrary delays, except for a guardedness restriction in (Sel-<:), which we discuss in Sec. 4.2. As such, delayed subtyping $\Gamma \vdash^{i} S <: T$ with $i = 0$ corresponds to ordinary subtyping $\Gamma \vdash S <: T$. In addition, the rules (Later-<:), (<:-Later) make it possible to push laters into the delay, and (<:-Add-Later) ensures that $\triangleright T$ is a supertype of T. While laters commute with intersection, union, and recursive types (as witnessed by the rules for type equality), pushing a later into the delay is needed in case rules like (Fld-<:-Fld), (Typ-<:-Typ) or ($\forall$-<:-$\forall$) are blocked by a later.

Term typing (rules (T-{}-E), (T-∀-E) are unchanged and elided) $\boxed{\Gamma \vdash e : T}$

$$\frac{\Gamma \vdash e : T_1 \qquad \Gamma \vdash^0 T_1 <: T_2}{\Gamma \vdash e : T_2}\ (\textsc{T-Sub}) \qquad \frac{\Gamma \vdash e : \triangleright T}{\Gamma \vdash \mathbf{coerce}\ e : T}\ (\textsc{T-Coerce})$$

$$\frac{\Gamma \vdash^0_{\mathsf{P}} p : T}{\Gamma \vdash p : T}\ (\textsc{T-Path}) \qquad \frac{\Gamma \mid x : \triangleright T \vdash \{\bar{d}\} : T}{\Gamma \vdash \nu x.\, \{\bar{d}\} : \mu x.\, T}\ (\textsc{T-\{\}-I})$$

$$\frac{\Gamma_1 \gg_{\triangleright} \Gamma_2 \qquad \Gamma_2, x : S \vdash e : T \qquad x \notin \mathsf{FV}(S)}{\Gamma_1 \vdash \lambda x.\, e : \forall x : S.\, T}\ (\textsc{T-∀-I-Strong}) \qquad \frac{\Gamma \vdash e : \forall z : S.\, T \qquad \Gamma \vdash^0_{\mathsf{P}} p : S}{\Gamma \vdash e\ p : T[z := p]}\ (\textsc{T-∀-E}_p)$$

Path typing (rule (P-∧-I) is derivable) $\boxed{\Gamma \vdash^{i}_{\mathsf{P}} p : T}$

$$\frac{x : T \in \Gamma}{\Gamma \vdash^0_{\mathsf{P}} x : T}\ (\textsc{P-Var}) \qquad \frac{\Gamma \vdash^i_{\mathsf{P}} p : T[z := p]}{\Gamma \vdash^i_{\mathsf{P}} p : \mu z.\, T}\ (\textsc{P-}\mu\textsc{-I}) \qquad \frac{\Gamma \vdash^i_{\mathsf{P}} p : \mu z.\, T}{\Gamma \vdash^i_{\mathsf{P}} p : T[z := p]}\ (\textsc{P-}\mu\textsc{-E})$$

$$\frac{\Gamma \vdash^i_{\mathsf{P}} p : T \qquad \Gamma \vdash^i T <: U}{\Gamma \vdash^i_{\mathsf{P}} p : U}\ (\textsc{P-Sub}) \qquad \frac{\Gamma \vdash^i_{\mathsf{P}} p : \{\mathsf{a} : T\}}{\Gamma \vdash^i_{\mathsf{P}} p.\mathsf{a} : T}\ (\textsc{P-Fld-E}) \qquad \frac{\Gamma \vdash^i_{\mathsf{P}} p.\mathsf{a} : T}{\Gamma \vdash^i_{\mathsf{P}} p : \{\mathsf{a} : T\}}\ (\textsc{P-Fld-I})$$

$$\frac{\Gamma \vdash^i_{\mathsf{P}} p : q.\mathbf{type} \qquad \Gamma \vdash^i_{\mathsf{P}} q : T}{\Gamma \vdash^i_{\mathsf{P}} p : T}\ (\textsc{P-Sngl-Trans}) \qquad \frac{\Gamma \vdash^i_{\mathsf{P}} p : q.\mathbf{type} \qquad \Gamma \vdash^i_{\mathsf{P}} q.\mathsf{a} : T}{\Gamma \vdash^i_{\mathsf{P}} p.\mathsf{a} : q.\mathsf{a}.\mathbf{type}}\ (\textsc{P-Sngl-E})$$

$$\frac{\Gamma \vdash^i_{\mathsf{P}} p : T}{\Gamma \vdash^i_{\mathsf{P}} p : p.\mathbf{type}}\ (\textsc{P-Sngl-Refl}) \qquad \frac{\Gamma \vdash^i_{\mathsf{P}} p : q.\mathbf{type}}{\Gamma \vdash^i_{\mathsf{P}} q : \top}\ (\textsc{P-Sngl-Inv}) \qquad \frac{\Gamma \vdash^i_{\mathsf{P}} p : \triangleright T}{\Gamma \vdash^{i+1}_{\mathsf{P}} p : T}\ (\textsc{P-Later})$$

Definition typing (rule (D-Path-Sngl) is derivable, (D-And) unchanged and elided) $\boxed{\Gamma \mid x : V \vdash \{\bar{d}\} : T}$

$$\frac{\Gamma, x : V \vdash v : T}{\Gamma \mid x : V \vdash \{\mathsf{a} = v\} : \{\mathsf{a} : T\}}\ (\textsc{D-Val}) \qquad \frac{\Gamma, x : V \mid z : x.\mathsf{a}.\mathbf{type} \wedge \triangleright T \vdash \{\bar{d}\} : T}{\Gamma \mid x : V \vdash \{\mathsf{a} = \nu z.\, \{\bar{d}\}\} : \{\mathsf{a} : \mu z.\, T\}}\ (\textsc{D-Val-New})$$

$$\frac{\Gamma, x : V \vdash^0 \triangleright L <: \triangleright T \qquad \Gamma, x : V \vdash^0 \triangleright T <: \triangleright U}{\Gamma \mid x : V \vdash \{\mathsf{A} = T\} : \{\mathsf{A} >: L <: U\}}\ (\textsc{D-Typ-Abs}) \qquad \frac{\Gamma, x : V \vdash^0_{\mathsf{P}} p : T}{\Gamma \mid x : V \vdash \{\mathsf{a} = p\} : \{\mathsf{a} : T\}}\ (\textsc{D-Path})$$

$$\frac{\Gamma \mid x : V \vdash \{\mathsf{a} = p\} : \{\mathsf{a} : T_1\} \qquad \Gamma, x : V \vdash^0 T_1 <: T_2}{\Gamma \mid x : V \vdash \{\mathsf{a} = p\} : \{\mathsf{a} : T_2\}}\ (\textsc{D-Path-Sub})$$

Notable derivable typing rules

$$\frac{\Gamma \vdash e : p.\mathsf{A} \qquad \Gamma \vdash^i_{\mathsf{P}} p : \{\mathsf{A} >: L <: U\}}{\Gamma \vdash \mathbf{coerce}^{i+1}\ e : U}\ (\textsc{T-Sel-Unfold}) \qquad \frac{\Gamma, x : S \vdash e : T \qquad x \notin \mathsf{FV}(S)}{\Gamma \vdash \lambda x.\, e : \forall x : S.\, T}\ (\textsc{T-∀-I})$$

$$\frac{\Gamma \vdash^i_{\mathsf{P}} p : T_1 \qquad \Gamma \vdash^i_{\mathsf{P}} p : T_2}{\Gamma \vdash^i_{\mathsf{P}} p : T_1 \wedge T_2}\ (\textsc{P-∧-I}) \qquad \frac{\Gamma \vdash^i_{\mathsf{P}} p : q.\mathbf{type}}{\Gamma \vdash^i_{\mathsf{P}} q : p.\mathbf{type}}\ (\textsc{P-Sngl-Sym})$$

$$\frac{\Gamma, x : V \vdash_{\mathsf{P}} p : T}{\Gamma \mid x : V \vdash \{\mathsf{a} = p\} : \{\mathsf{a} : p.\mathbf{type}\}}\ (\textsc{D-Path-Sngl}) \qquad \frac{\Gamma, x : V, z : S \vdash t : T \qquad z \notin \mathsf{FV}(S)}{\Gamma \mid x : \triangleright V \vdash \{\mathsf{a} = \lambda z.\, t\} : \{\mathsf{a} : \forall z : S.\, T\}}\ (\textsc{D-∀})$$

Fig. 6. gDOT rules for term typing, path typing, and definition typing.

Type equality (equivalence and congruence rules are omitted) $\boxed{T_1 \equiv T_2}$

$$\triangleright (T_1 \wedge T_2) \equiv \triangleright T_1 \wedge \triangleright T_2 \qquad \triangleright (T_1 \vee T_2) \equiv \triangleright T_1 \vee \triangleright T_2 \qquad \triangleright (\mu x.\, T) \equiv \mu x.\, \triangleright T$$

Bounded, distributive subtyping lattice $\boxed{\Gamma \vdash^{i} T_1 <: T_2}$

$$\Gamma \vdash^{i} T <: \top \;(\text{<:-}\top) \qquad \Gamma \vdash^{i} T_1 \wedge T_2 <: T_1 \;(\wedge_1\text{-<:}) \qquad \Gamma \vdash^{i} T_1 \wedge T_2 <: T_2 \;(\wedge_2\text{-<:}) \qquad \Gamma \vdash^{i} \bot <: T \;(\bot\text{-<:})$$

$$\frac{\Gamma \vdash^{i} T <: U_1 \qquad \Gamma \vdash^{i} T <: U_2}{\Gamma \vdash^{i} T <: U_1 \wedge U_2}\;(\text{<:-}\wedge) \qquad \Gamma \vdash^{i} T <: T \;(\text{<:-}\textsc{Refl}) \qquad \frac{\Gamma \vdash^{i} S <: T \qquad \Gamma \vdash^{i} T <: U}{\Gamma \vdash^{i} S <: U}\;(\text{<:-}\textsc{Trans})$$

$$\Gamma \vdash^{i} T_1 <: T_1 \vee T_2 \;(\text{<:-}\vee_1) \qquad \Gamma \vdash^{i} T_2 <: T_1 \vee T_2 \;(\text{<:-}\vee_2) \qquad \frac{\Gamma \vdash^{i} T_1 <: U \qquad \Gamma \vdash^{i} T_2 <: U}{\Gamma \vdash^{i} T_1 \vee T_2 <: U}\;(\vee\text{-<:})$$

$$\Gamma \vdash^{i} (S \vee T) \wedge U <: (S \wedge U) \vee (T \wedge U) \;(\textsc{Distr}\text{-}\wedge\text{-}\vee\text{-<:}) \qquad \frac{T_1 \equiv T_2}{\Gamma \vdash^{i} T_1 <: T_2}\;(\text{<:-}\textsc{Eq})$$

Later types

$$\frac{\Gamma \vdash^{i+1} T <: U}{\Gamma \vdash^{i} \triangleright T <: \triangleright U}\;(\textsc{Later}\text{-<:}) \qquad \frac{\Gamma \vdash^{i} \triangleright T <: \triangleright U}{\Gamma \vdash^{i+1} T <: U}\;(\text{<:-}\textsc{Later}) \qquad \Gamma \vdash^{i} T <: \triangleright T \;(\text{<:-}\textsc{Add-Later})$$

Type members

$$\frac{\Gamma \vdash^{i}_{\mathsf{p}} p : \{\mathsf{A} >: L <: U\}}{\Gamma \vdash^{i} \triangleright L <: p.\mathsf{A}}\;(\text{<:-}\textsc{Sel}) \qquad \frac{\Gamma \vdash^{i}_{\mathsf{p}} p : \{\mathsf{A} >: L <: U\}}{\Gamma \vdash^{i} p.\mathsf{A} <: \triangleright U}\;(\textsc{Sel}\text{-<:})$$

Recursive types

$$\frac{\Gamma, x : \triangleright^{i} T_1 \vdash^{i} T_1 <: T_2}{\Gamma \vdash^{i} \mu x.\, T_1 <: \mu x.\, T_2}\;(\mu\text{-<:-}\mu) \qquad \frac{x \notin T}{\Gamma \vdash^{i} \mu x.\, T <: T}\;(\mu\text{-<:}) \qquad \frac{x \notin T}{\Gamma \vdash^{i} T <: \mu x.\, T}\;(\text{<:-}\mu)$$

Co/contra-variant subtyping

$$\frac{\Gamma \vdash^{i} T_1 <: T_2}{\Gamma \vdash^{i} \{\mathsf{a} : T_1\} <: \{\mathsf{a} : T_2\}}\;(\textsc{Fld}\text{-<:-}\textsc{Fld}) \qquad \frac{\Gamma \vdash^{i} \triangleright L_2 <: \triangleright L_1 \qquad \Gamma \vdash^{i} \triangleright U_1 <: \triangleright U_2}{\Gamma \vdash^{i} \{\mathsf{A} >: L_1 <: U_1\} <: \{\mathsf{A} >: L_2 <: U_2\}}\;(\textsc{Typ}\text{-<:-}\textsc{Typ})$$

$$\frac{\Gamma \vdash^{i} \triangleright S_2 <: \triangleright S_1 \qquad \Gamma, x : \triangleright^{i+1} T_2 \vdash^{i} \triangleright T_1 <: \triangleright T_2}{\Gamma \vdash^{i} \forall x : S_1.\, T_1 <: \forall x : S_2.\, T_2}\;(\forall\text{-<:-}\forall)$$

Singleton types (rule ($\textsc{Sngl}_{qp}$-<:) is derivable and thus elided)

$$\frac{\Gamma \vdash^{i}_{\mathsf{p}} p : q.\mathbf{type} \qquad T_1 \cong^{*}_{p:=q} T_2}{\Gamma \vdash^{i} T_1 <: T_2}\;(\textsc{Sngl}_{pq}\text{-<:}) \qquad \frac{\Gamma \vdash^{i}_{\mathsf{p}} p : T \qquad \Gamma \vdash^{i} p.\mathbf{type} <: q.\mathbf{type}}{\Gamma \vdash^{i} q.\mathbf{type} <: p.\mathbf{type}}\;(\textsc{Sngl}\text{-<:-}\textsc{Sym})$$

$$\frac{\Gamma \vdash^{i}_{\mathsf{p}} p : T}{\Gamma \vdash^{i} p.\mathbf{type} <: T}\;(\textsc{Sngl}\text{-<:-}\textsc{Self})$$

Notable derivable subtyping rules

$$\frac{\Gamma, x : \triangleright^{i} T_1 \vdash^{i} T_1 <: T_2}{\Gamma \vdash^{i} \mu x.\, T_1 <: T_2}\;(\textsc{Bind-1}) \qquad \frac{\Gamma, x : \triangleright^{i} T_1 \vdash^{i} T_1 <: T_2}{\Gamma \vdash^{i} T_1 <: \mu x.\, T_2}\;(\textsc{Bind-2})$$

Fig. 7. gDOT rules for type equality and subtyping.

4.2 Type Selections

The rules (<:-Sel) and (Sel-<:) for selectors derive $\Gamma \vdash^i\ \triangleright L <: p.\mathsf{A}$ and $\Gamma \vdash^i p.\mathsf{A} <: \triangleright U$ from $\Gamma \vdash^i_{\mathsf{p}} p : \{\mathsf{A} >: L <: U\}$. These rules include a guardedness restriction in the form of a later, as imposed by our semantic model. Intuitively, the model imposes this restriction because semantic types occur under a later operator (►) in the recursive domain equation (Sec. 1.3).

The presence of the later makes rule (Sel-<:) of gDOT weaker than the corresponding rule in pDOT, but luckily we we can adapt programs by inserting laters. The way to eliminate them depends on whether we consider the path or term judgment. In path typing, laters are eliminated by increasing the path-typing delay, *e.g.*:

$$\cfrac{\cfrac{\Gamma \vdash^i_{\mathsf{p}} p : y.\mathsf{B} \qquad \Gamma \vdash^i_{\mathsf{p}} y : \{\mathsf{B} >: \bot <: \{\mathsf{A} >: L <: U\}\}}{\Gamma \vdash^i_{\mathsf{p}} p : \triangleright \{\mathsf{A} >: L <: U\}}\ \text{(Sel-<:,P-Sub)}}{\Gamma \vdash^{i+1}_{\mathsf{p}} p : \{\mathsf{A} >: L <: U\}}\ \text{(P-Later)}$$

Delayed path typings can be used as premises of subtyping judgments for path selections with rules (<:-Sel,Sel-<:), which in turn can be used for subsumption of both paths and terms with rules (P-Sub,T-Sub), resulting in the accumulation of delays. One can also defer eliminating laters, and use rule (T-Path) to obtain a term typing judgment whose type involves laters.

In term typing, laters are eliminated using coercions, thanks to rule (T-Coerce) and derived rule (T-Sel-Unfold). Continuing our derivation above, and inlining the derivation of (T-Sel-Unfold), we have the following derivation:

$$\cfrac{\Gamma \vdash e : p.\mathsf{A} \qquad \cfrac{\cfrac{}{\Gamma \vdash^0 p.\mathsf{A} <: \triangleright^{i+1} p.\mathsf{A}}\ \text{(<:-Add-Later)} \qquad \cfrac{\cfrac{\cfrac{\Gamma \vdash^{i+1}_{\mathsf{p}} p : \{\mathsf{A} >: L <: U\}}{\Gamma \vdash^{i+1} p.\mathsf{A} <: \triangleright U}\ \text{(Sel-<:)}}{\Gamma \vdash^0 \triangleright^{i+1} p.\mathsf{A} <: \triangleright^{i+2} U}\ \text{(Later-<:)}}{}}{\Gamma \vdash^0 p.\mathsf{A} <: \triangleright^{i+2} U}\ \text{(<:-Trans)}}{\Gamma \vdash \mathbf{coerce}^{i+2}\ e : U}\ \text{(T-Sub, T-Coerce)}$$

We believe all pDOT programs can be adapted to gDOT in this fashion, as discussed in Sec. 9.

Dual to the rules (<:-Sel) and (Sel-<:), which only give the bounds L and U of $p : \{\mathsf{A} >: L <: U\}$ under a later, the rule (D-Typ-Abs) for introduction of type members $\{\mathsf{A} = T\} : \{\mathsf{A} >: L <: U\}$ only requires subtyping of T with respect to bounds L and U under a later.

4.3 Function Introduction

The rule (T-∀-I-Strong) enables eliminating a later from each variable in the context when introducing a function. At the core of this rule we find the judgment $\Gamma_1 \gg_{\triangleright} \Gamma_2$, which is defined as the element-wise reflexive congruence closure under $\triangleright$, $\wedge$ and $\vee$ of $\triangleright T \gg_{\triangleright} T$. While rule (D-∀) is a common special case of (T-∀-I-Strong), stripping a later from the typing contexts created by (D-Val-New) requires the additional generality of (T-∀-I-Strong).

4.4 Other Typing Rules

Distributivity. Rule (Distr-∧-∨-<:), together with derived rules, *e.g.*, the dual of (Distr-∧-∨-<:), make gDOT's subtyping lattice *distributive*, helping to deal with the interaction of intersection and union types. This rule revealed itself necessary in Sec. 6.3. Other typing rules, which help distribute certain type constructors over each other, are left to our appendix [Giarrusso et al. 2020].

Recursive Types. gDOT supports subtyping for recursive types [Rompf and Amin 2016], via rule (μ-<:-μ) (*cf.* Rompf and Amin's rule (BindX)), and allows one to drop unused μ binders via rules (μ-<:) and (<:-μ). The latter rules enable one to derive Rompf and Amin's rule (Bind1) and their conjectured rule (Bind2). These rules are absent from WadlerFest DOT and pDOT, requiring the insertion of redundant let bindings in some programs.

Singleton Types. gDOT has various rules on path aliasing that are supported by Scala compilers but not derivable (to the best of our knowledge) in pDOT:

- Aliasing is reflexive, *i.e.,* any well-typed path aliases itself (P-Sngl-Refl),
- Aliasing is symmetric (Sngl-<:-Sym).
- If p aliases q, *i.e.,* $\Gamma \vdash^{i}_{\mathsf{p}} p : q.\mathbf{type}$, then q is well-typed (P-Sngl-Inv).
- Singleton type $p.\mathbf{type}$ is a subtype of any type T of p (Sngl-<:-Self).

Rules (P-Sngl-Refl) and (Sngl-<:-Self) enable deriving pDOT's primitive rule (P-∧-I). Using all four rules we derive (P-Sngl-Sym), and in turn pDOT's primitive rule (Sngl$_{qp}$-<:).

5 SEMANTIC SOUNDNESS

We define a semantic model using the technique of logical relations to prove type soundness of gDOT — well-typed terms do not go wrong, *i.e.,* they are safe in the following sense:

Definition 5.1 (Safety). *A term e is* safe *if, for all e' such that $e \rightarrow^{*}_{\mathsf{t}} e'$, term e' is not stuck, that is, either e' is a value or e' can reduce.*

Theorem 5.2 (Type soundness). *If $\varepsilon \vdash e : T$, then e is safe.*

The main ingredient of a semantic soundness proof is the semantic typing judgment $\Gamma \vDash e : T$, which expresses what programs are safe in terms of their *behavior.* The semantic typing judgment is different from and more flexible than the syntactic typing judgment $\Gamma \vdash e : T$, which is defined inductively and dictates what programs are safe using a fixed set of rules. First, we can prove each typing rule as a lemma. For example, the rule (D-Val) becomes the lemma:

$$\Gamma, x : V \vDash v : T \qquad \text{implies} \qquad \Gamma \mid x : V \vDash \{\mathsf{a} = v\} : \{\mathsf{a} : T\}$$

Stating typing rules as lemmas on a semantic model has a tangible benefit — it enables varying existing rules and experimenting with new rules. The semantic model will suggest necessary restrictions or generalizations to make these rules sound. This is how we designed many of the new typing rules of gDOT in Sec. 4.

Second, beyond proving typing lemmas, we can prove semantic typing judgments for programs that are not syntactically well-typed, *e.g.,* for programs that make use of unsafe casts whose correctness relies on (g)DOT's support for data abstraction. In Sec. 6 we give examples of proofs for such programs, including the function `typeFromTypeRefUnsafe` from Sec. 1.1.

As explained in Sec. 1.3, to define the semantic typing judgment $\Gamma \vDash e : T$, one needs to define a mapping from syntactic types T to *semantic types* $\mathcal{V}[\![T]\!] \in \mathsf{SemType}$, which express what values are safe for a given type T. And to do that, we must define the semantic domain of types SemType. To model impredicative type members in (g)DOT, our SemType is defined using *step-indexing* as the solution to the recursive domain equation Eq. (Domain) (ignoring paths and primitives):

$$\mathsf{SemType} \triangleq (\mathsf{Var} \rightarrow \mathsf{SemVal}) \rightarrow \mathsf{SemVal} \rightarrow \mathsf{iProp}$$
$$\mathsf{SemVal} \cong \mathsf{Var} + \left\{ \lambda x.\, e \right\} + (\mathsf{Label} \xrightarrow{\text{fin}} (\blacktriangleright \mathsf{SemType} + \mathsf{SemVal}))$$

This recursive domain equation could *in principle* be solved directly in Iris. However, we will solve it *indirectly.* Instead of letting values contain semantic types, we let values contain so-called *stamps.* In turn, stamps are mapped to semantic types through Iris's machinery for *saved predicates* [Jung et al. 2016]. As an additional benefit, this allows us to keep the syntax of values first-order, which aids mechanization in Coq. In the rest of this section, we describe gDOT with stamps — called *stamped gDOT* — in Sec. 5.1, the Iris logic in Sec. 5.2, and finally our semantic model in Sec. 5.3.

$$\textsc{$\triangleright$-Intro}\quad P \vdash_{\mathrm{I}} \triangleright P \qquad \textsc{$\triangleright$-Mono}\quad \frac{P \vdash_{\mathrm{I}} Q}{\triangleright P \vdash_{\mathrm{I}} \triangleright Q} \qquad \textsc{$\triangleright$-Impl}\quad \triangleright(P \Rightarrow Q) \vdash_{\mathrm{I}} (\triangleright P \Rightarrow \triangleright Q) \qquad \textsc{Impl-$\triangleright$}\quad (\triangleright P \Rightarrow \triangleright Q) \vdash_{\mathrm{I}} \triangleright(P \Rightarrow Q)$$

$$\textsc{Löb}\quad (\triangleright P \Rightarrow P) \vdash_{\mathrm{I}} P \qquad \textsc{$|\!\dot{\Rrightarrow}$-mono}\quad \frac{P \vdash_{\mathrm{I}} Q}{|\!\dot{\Rrightarrow} P \vdash_{\mathrm{I}} |\!\dot{\Rrightarrow} Q} \qquad \textsc{$|\!\dot{\Rrightarrow}$-intro}\quad P \vdash_{\mathrm{I}} |\!\dot{\Rrightarrow} P \qquad \textsc{$|\!\dot{\Rrightarrow}$-trans}\quad |\!\dot{\Rrightarrow} |\!\dot{\Rrightarrow} P \vdash_{\mathrm{I}} |\!\dot{\Rrightarrow} P \qquad \textsc{$|\!\dot{\Rrightarrow}$-frame}\quad Q \land |\!\dot{\Rrightarrow} P \vdash_{\mathrm{I}} |\!\dot{\Rrightarrow}(Q \land P)$$

$$\textsc{Saved-Pred-Alloc}\quad \mathsf{True} \vdash_{\mathrm{I}} |\!\dot{\Rrightarrow} \exists s.\, (s \rightsquigarrow \varphi) \qquad \textsc{Saved-Pred-Agree}\quad (s \rightsquigarrow \varphi_1) \land (s \rightsquigarrow \varphi_2) \vdash_{\mathrm{I}} \triangleright(\varphi_1 = \varphi_2)$$

Fig. 8. A selection of proof rules of the considered fragment of Iris.

5.1 Stamped gDOT

We introduce *stamped gDOT*, in which values refer to semantic types *indirectly* through stamps, as discussed in Sec. 5. To disambiguate between *unstamped gDOT* (which we have used until now) and stamped gDOT, in this section we color the syntax of unstamped gDOT in blue, while keeping that of stamped gDOT in black (or red for emphasis). Unstamped and stamped gDOT share their syntax *except* for definition bodies, which are respectively:

$$\mathsf{DefBody} \ni d ::= p \mid T \qquad\qquad \mathsf{DefBody} \ni d ::= p \mid \sigma, s$$

Unstamped and stamped types coincide, because types cannot contain definitions. In stamped gDOT, *stamps* $s \in \mathsf{Stamp}$ are simply identifiers, while *deferred substitutions* $\sigma \in \mathsf{Subst} \triangleq \mathsf{Var} \xrightarrow{\text{fin}} \mathsf{Val}$ start as identity substitutions and accumulate substitutions applied to the surrounding value.

The operational semantics of stamped gDOT also resembles that of unstamped gDOT. While type members do not affect the operational semantics of either gDOT version, substitutions affect type members — either directly, in unstamped gDOT, or indirectly though the deferred substitution σ, in stamped gDOT. This becomes particularly relevant in gDOT's semantic model (Sec. 5.3).

To relate unstamped and stamped gDOT, we define the relation $e \approx e'$, which expresses that an unstamped gDOT term e and a stamped gDOT term e' are equal modulo type members. Since type members do not affect the operational semantics, $e \approx e'$ implies that e is safe if and only if e' is safe. This property is crucial to prove adequacy of semantic typing (Theorem 5.4).

5.2 The Iris Logic

The Iris framework provides a programming-language-independent step-indexed separation logic, which we instantiate with the stamped gDOT language. Since (stamped) gDOT is a pure language, we do not use Iris's separating conjunction ($*$) and magic wand ($-\!\!*$), but use ordinary conjunction ($\land$) and implication ($\Rightarrow$) everywhere.[6] Concretely, we use the following fragment of Iris:

$$\begin{aligned} \tau &::= \mathbf{0} \mid \mathbf{1} \mid \mathsf{iProp} \mid \blacktriangleright\tau \mid \tau + \tau \mid \tau \times \tau \mid \tau \to \tau \mid \mathsf{Term} \mid \mathsf{Val} \mid \ldots \\ t, u, P, Q, \varphi &::= x \mid \lambda x : \tau.\, t \mid t(u) \mid \mathsf{False} \mid \mathsf{True} \mid t =_\tau u \mid P \land Q \mid P \lor Q \mid P \Rightarrow Q \\ &\quad\mid \exists x : \tau.\ P \mid \forall x : \tau.\ P \mid \triangleright P \mid \mu x : \tau.\ t \mid s \rightsquigarrow \varphi \mid |\!\dot{\Rrightarrow} P \mid \ldots \end{aligned}$$

Since Iris is a higher-order logic, its grammar includes the simply-typed lambda-calculus with a number of primitive types and terms operating on these types. Most important is the type iProp of

[6] By restricting Iris's ghost state mechanism to saved predicates, all Iris propositions we consider are persistent. Hence, no persistence modality is needed, and separating conjunction/magic wand and ordinary conjunction/implication coincide.

Iris propositions, and the types for all syntactical categories of gDOT (*e.g.,* Term and Val). The Iris typing judgment is mostly standard and can be derived from the use of meta variables — τ ranges over Iris types, P and Q range over Iris propositions, and t and u range over Iris terms of any type. The fragment of Iris we consider includes:

- The *higher-order and impredicative quantifiers* $\exists\, x : \tau.\ P$ and $\forall\, x : \tau.\ P$, which allow us to quantify over semantic types (*i.e.,* Iris predicates) in our semantic model.
- The *later modality* $\triangleright$ for step-indexing and the *guarded fixpoint* operator $\mu\, x : \tau.\ t$ (Sec. 5.2.1).
- The connective for *saved predicates* $s \rightsquigarrow \varphi$ and the *update modality* $\mathrel{|\!\!\dot{\Rrightarrow}}$, which are used to assign semantic types to stamps (Sec. 5.2.4).

We write $P \vdash_{\mathrm{I}} Q$ when P entails Q in Iris. Selected Iris rules are shown in Fig. 8. Our Iris fragment (with saved predicates, not arbitrary ghost state) enjoys the rule (Impl-$\triangleright$), which does not hold in full Iris. We need this rule to prove (among others) the semantic typing lemmas for contravariant subtyping rules (Typ-<:-Typ) and ($\forall$-<:-$\forall$).

5.2.1 Abstract Step-Indexing. The purpose of step-indexing is to stratify circular definitions using a natural number, called the *step-index*. While circular definitions are indexed by an explicit step-index in traditional step-indexed models [Ahmed 2004; Appel and McAllester 2001], Iris hides step-indices using an abstract form of step-indexing [Appel et al. 2007; Birkedal et al. 2011], which implicitly indexes logical propositions with a step-index. The later modality ($\triangleright$) [Nakano 2000] can then be used to reason about step-indexing internally — $\triangleright P$ asserts that P holds when the step-index is decremented (*i.e.,* it holds one step *later*). As we will see in Sec. 5.2.4, the later modality is crucial to soundly support saved predicates. In addition, it gives rise to Löb induction (Löb), which says that to prove P, we can prove P under the assumption $\triangleright P$. Löb induction implicitly performs induction on the number of steps. Moreover, the guarded fixpoint operator $\mu\, x : \tau.\ t$ can be used to construct recursive definitions with no restrictions on the variance of the recursive occurrence x in t, but requires x to be guarded, *i.e.,* to appear under a later.

5.2.2 Term Weakest Preconditions. Support for reasoning about programs (using weakest preconditions) is not hard-wired into Iris. Instead, one can define custom reasoning principles for any chosen programming language [Krebbers et al. 2017a]. Since gDOT is pure and deterministic, we define a custom notion of pure weakest preconditions for stamped gDOT terms:

$$\mathsf{wp}\ e\ \{\varphi\} \triangleq \begin{cases} \varphi(e) & \text{if } e \in \mathsf{Val} \\ \exists\, e'.\, (e \longrightarrow_{\mathrm{t}} e') \wedge \triangleright\, \mathsf{wp}\ e'\ \{\varphi\} & \text{otherwise} \end{cases}$$

Intuitively, $\mathsf{wp}\ e\ \{\varphi\}$ asserts that e is safe, and any resulting value v of e satisfies $\varphi(v)$. We write $\mathsf{wp}\ e\ \{v.\ P\}$ as shorthand for $\mathsf{wp}\ e\ \{\lambda\, v.\ P\}$.

Like Iris's standard weakest precondition for stateful languages, we use Iris's guarded fixpoint operator $\mu x : \tau. t$. The later modality in the inductive case connects Iris's abstract step-indexing to the physical computation steps of the program — it introduces a later for each computation step. For example, this is crucial for proving partial program correctness using (Löb) induction, which allows proving P while assuming the induction hypothesis $\triangleright P$. By unfolding the weakest precondition for one computation step, one can turn the induction hypothesis $\triangleright P$ into just P using ($\triangleright$-Mono).

5.2.3 Path Weakest Preconditions. While weakest preconditions for terms ensure partial correctness, weakest preconditions for paths ensure total correctness (*i.e.,* normalization):

$$\mathsf{wp}_{\mathsf{P}}\ p\ \{\varphi\} \triangleq \begin{cases} \varphi(p) & \text{if } p \in \mathsf{Val} \\ \exists\, v_q, q'.\ \mathsf{wp}_{\mathsf{P}}\ q\ \{v.\ v = v_q\} \wedge v_q.\mathsf{a} \searrow q' \wedge \mathsf{wp}_{\mathsf{P}}\ q'\ \{\varphi\} & \text{if } p = q.\mathsf{a} \end{cases}$$

Intuitively, $\mathsf{wp}_{\mathsf{P}}\ p\ \{\varphi\}$ asserts that p normalizes to a value v satisfying $\varphi(v)$. We write $\mathsf{wp}_{\mathsf{P}}\ p\ \{v.\ P\}$ as shorthand for $\mathsf{wp}_{\mathsf{P}}\ p\ \{\lambda\, v.\ P\}$.

The connective for path weakest preconditions is rather different from the version for terms — it does not include a later modality ($\triangleright$), and it is thus formalized as a least fixpoint instead of a guarded fixpoint. The absence of the later modality ensures that paths normalize. This is crucial because types in DOT can depend on paths, and thus each path is required to denote a unique value. While dropping the later modality is essential for normalization, it comes at a price. Unlike term weakest preconditions, path weakest preconditions do not introduce a later (and thereby strip off a later of each hypotheses) when performing a step of computation. Hence, adding a **coerce** construct to paths would not be helpful, as it would not allow stripping off laters. This restriction explains why we index the path typing judgment with a delay instead. The definition of path weakest preconditions is inspired by the definition of *total* weakest preconditions in Iris. However, while Iris's definition is tailored to a stateful language, ours is tailored to paths, which are pure and deterministic.

5.2.4 Saved Predicates. To represent the mapping from stamps to semantic types in our semantic model (Sec. 5.3), we use saved predicates, which are an instance of Iris's machinery for higher-order ghost state [Jung et al. 2016]. The saved predicate connective $s \rightsquigarrow \varphi$ assert that stamp identifier s points to Iris predicate φ. They enjoy the rule (Saved-Pred-Agree): if an identifier maps to two predicates, they are equal. This equality appears under a later modality ($\triangleright$) because saved predicates can refer to themselves, and such self-references must be guarded through a later modality ($\triangleright$) to be sound [Jung et al. 2018b, Sec. 3.3].

Saved predicates are allocated via rule (Saved-Pred-Alloc), which allows one to obtain $s \rightsquigarrow \varphi$ for a given Iris predicate φ. The allocation rule involves Iris's *basic update modality* $|\!\dot{\Rrightarrow} P$, a strong monad that is used to modify Iris's ghost state. Readers that are unfamiliar with Iris can gloss over this modality.

In our semantic model, we instantiate Iris's generic saved predicate construction with semantic types SemType, *i.e.,* predicates over stamped gDOT environments and stamped gDOT values:

$$\mathsf{SemType} \triangleq (\mathsf{Var} \to \mathsf{Val}) \to \mathsf{Val} \to \mathsf{iProp}$$

To see how the use of saved predicates relates to the explicit model as a solution of the recursive domain equation Eq. (Domain) at page 6, let us unfold the model of Iris. The type of propositions of (our fragment of) Iris is the solution to the following recursive domain equation:

$$\mathsf{iProp} \cong (\mathsf{Stamp} \xrightarrow{\text{fin}} \text{\textsc{Ag}}(\blacktriangleright \mathsf{SemType})) \to \mathsf{siProp}$$

Here, siProp is the type of step-indexed propositions, and Ag is Iris's agreement camera, used to ensure that saved predicates are persistent. Most of these details can be ignored; what matters is that SemType, which contains a recursive occurrence of iProp, appears under a later type former ($\blacktriangleright$). As such, the model of Iris with saved predicates is isomorphic (modulo the indirection via stamps) to a direct model of gDOT as described by Eq. (Domain). Moreover, the later type ($\blacktriangleright$) in this equation explains the later modality ($\triangleright$) in the conclusion of rule (Saved-Pred-Agree).

5.3 The Semantic Model of gDOT

We now put Iris and stamped gDOT to work by proving type soundness of gDOT (Theorem 5.2). In Sec. 5.3.1 we define a semantic typing judgment $\Gamma \vDash e : T$ for stamped gDOT, which we lift in Sec. 5.3.2 to a semantic typing judgment $\Gamma \vDash e : T$ for unstamped gDOT. In Sec. 5.3.3 we prove that syntactically well-typed unstamped gDOT terms are also semantically well-typed:

Theorem 5.3 (Fundamental). *If* $\Gamma \vdash e : T$, *then* $\Gamma \vDash e : T$.

Finally, in Sec. 5.3.4 we prove adequacy of semantic typing for unstamped gDOT:

Theorem 5.4 (Adequacy). *If $\varepsilon \vDash e : T$, then e is safe.*

Combining these theorems shows type soundness (Theorem 5.2), *i.e.,* if $\varepsilon \vdash e : T$, then e is safe.

5.3.1 Stamped Semantic Typing Judgments. Fig. 9 shows our semantic model for stamped gDOT. Its definition follows the conventional setup of a logical relations model. First we define interpretation relations $\overline{\mathcal{D}}[\![T]\!]_\rho(\overline{d})$, $\mathcal{V}[\![T]\!]_\rho(v)$, and $\mathcal{E}[\![T]\!]_\rho(e)$ that describe the closed definition lists $\overline{d}$, closed values v, and closed terms e that safety inhabit a type T, under an environment $\rho \in \mathsf{Env} \triangleq \mathsf{Var} \longrightarrow \mathsf{Val}$ that gives the interpretation of the variables in T. These interpretation relations are defined by structural recursion on types T. Second, we lift these interpretation relations using closing substitutions to the various semantic typing judgments for open terms.

The value interpretations of the basic types are standard for a logical relations model. The interpretations of $\bot$, $\top$, $\land$, $\lor$ and $\triangleright$ use the corresponding logical connectives of Iris.

The value interpretation of function types $\mathcal{V}[\![\forall\,(x : S).\,T]\!]_\rho(v)$ expresses that v is α-equivalent to a function $\lambda x.\,e$ that maps values w of type S into terms $e[x := w]$ of type T. The latter is expressed by the term interpretation $\mathcal{E}[\![T]\!]_{(\rho,x:=w)}(e[x := w])$, which is defined using weakest preconditions as is standard for logical relations in Iris. However, since gDOT supports dependent functions (where the argument x is in scope in type T), we interpret T in the extended context $\rho, x := w$. Moreover, we use the later modality ($\triangleright$), like in step-indexed logical relations for *equi-recursive* types, where type constructors must be *contractive* rather than *non-expansive* [Appel and McAllester 2001]. This choice provides stronger typing rules, and is for instance the reason why (T-∀-I-Strong) can strip a later ($\triangleright$) from gDOT's typing context.

The value interpretation of record types $\mathcal{V}[\![\{\mathsf{a} : U\}]\!]_\rho(v)$ and $\mathcal{V}[\![\{\mathsf{A} >: L <: U\}]\!]_\rho(v)$ expresses that v is α-equivalent to an object $\nu x.\,\{\overline{d}\}$ with value member a (or type member A) that enjoys the right property. The latter is expressed using the interpretation $\overline{\mathcal{D}}[\![\_]\!]_\rho(\overline{d})$ for definition lists $\overline{d}$.

To define the interpretation of type members, we first define the auxiliary definition $s \rightsquigarrow_\sigma \psi$, which uses Iris's saved predicates to express that stamp s and deferred substitution σ map to *closed semantic type* $\psi \in \mathsf{Val} \longrightarrow \mathsf{iProp}$. The interpretation of type members itself $\overline{\mathcal{D}}[\![\{\mathsf{A} >: L <: U\}]\!]_\rho(\overline{d})$ says that when looking up the label A in $\overline{d}$ we obtain a stamp s and deferred substitution σ that map to (closed) semantic type ψ, which is bound by $\mathcal{V}[\![L]\!]_\rho$ and $\mathcal{V}[\![U]\!]_\rho$. Since semantic types ψ stored in values are guarded through saved predicates, we only refer to ψ under the later modality ($\triangleright$). This definition prevents bad bounds by ensuring that ψ respects its bounds L and U. For instance, $\mathcal{V}[\![\{\mathsf{A} >: \top <: \bot\}]\!]_\rho(v)$ is false. Therefore, objects with bad bounds cannot be typed in the empty context, and unsound subtyping evidence cannot be constructed.

The value interpretation of abstract types $\mathcal{V}[\![p.\mathsf{A}]\!]_\rho(v)$ states that p normalizes to object w, which in field A holds stamp s and deferred substitution σ, that together refer to closed semantic type ψ. Further, it asserts that v, under the later modality ($\triangleright$), satisfies the semantic type ψ. Since (paths in) types can contain variables, this definition substitutes ρ in p, and then uses the path weakest precondition to reason about the resulting object w. Similarly, the interpretation of singleton types $\mathcal{V}[\![p\,.\mathbf{type}]\!]_\rho(v)$ normalizes p to w, and checks that v and w coincide.

The value interpretation of μ-types $\mathcal{V}[\![\mu(x : T)]\!]_\rho(v)$ interprets T in the extended environment $\rho, x := v$, matching the informal semantics.

Using the interpretation relations, we define the semantic typing judgments. For instance, the semantic term typing judgment $\Gamma \vDash_{\mathsf{s}} e : T$ asserts that e runs safely in any environment ρ matching Γ, and results in a value satisfying $\mathcal{V}[\![T]\!]$. The semantic path typing judgment $\Gamma \vDash^i_{\mathsf{p}} p : T$ is defined using path weakest preconditions, so it asserts that path p normalizes to a value satisfying $\mathcal{V}[\![T]\!]$.

Auxiliary definitions

$$s \rightsquigarrow_\sigma \psi \triangleq \exists \varphi.\ (s \rightsquigarrow \varphi) \land \triangleright(\psi = \varphi(\sigma))$$

Definition interpretation $\boxed{\overline{\mathcal{D}}[\![\_]\!]\_(\_) : \mathsf{Type} \to \mathsf{Env} \to \mathsf{DefList} \to \mathsf{iProp}}$

$$\begin{aligned}
\overline{\mathcal{D}}[\![\top]\!]_\rho(\bar{d}) &\triangleq \mathsf{True} \\
\overline{\mathcal{D}}[\![S \land T]\!]_\rho(\bar{d}) &\triangleq \overline{\mathcal{D}}[\![S]\!]_\rho(\bar{d}) \land \overline{\mathcal{D}}[\![T]\!]_\rho(\bar{d}) \\
\overline{\mathcal{D}}[\![\{\mathsf{a} : T\}]\!]_\rho(\bar{d}) &\triangleq \exists p.\ \mathsf{lookup}(\mathsf{a}, \bar{d}) = p \land \mathsf{wp_P}\ p\ \{\mathcal{V}[\![T]\!]_\rho\} \\
\overline{\mathcal{D}}[\![\{\mathsf{A} >: L <: U\}]\!]_\rho(\bar{d}) &\triangleq \exists \sigma, s, \psi.\ \mathsf{lookup}(\mathsf{A}, \bar{d}) = (\sigma, s) \land (s \rightsquigarrow_\sigma \psi) \land \\
&\qquad \big(\forall v.\ \triangleright \mathcal{V}[\![L]\!]_\rho(v) \Rightarrow \triangleright \psi(v)\big) \land \big(\forall v.\ \triangleright \psi(v) \Rightarrow \triangleright \mathcal{V}[\![U]\!]_\rho(v)\big) \\
\overline{\mathcal{D}}[\![T]\!]_\rho(\bar{d}) &\triangleq \mathsf{False} \qquad\qquad (\text{if } T \text{ is not } \top, S \land T, \{\mathsf{A} >: L <: U\}, \text{or } \{\mathsf{a} : T\})
\end{aligned}$$

Value interpretation $\boxed{\mathcal{V}[\![\_]\!]\_(\_) : \mathsf{Type} \to \mathsf{Env} \to \mathsf{Val} \to \mathsf{iProp}}$

$$\begin{aligned}
\mathcal{V}[\![\top]\!]_\rho(v) &\triangleq \mathsf{True} \\
\mathcal{V}[\![\bot]\!]_\rho(v) &\triangleq \mathsf{False} \\
\mathcal{V}[\![S \land T]\!]_\rho(v) &\triangleq \mathcal{V}[\![S]\!]_\rho(v) \land \mathcal{V}[\![T]\!]_\rho(v) \\
\mathcal{V}[\![S \lor T]\!]_\rho(v) &\triangleq \mathcal{V}[\![S]\!]_\rho(v) \lor \mathcal{V}[\![T]\!]_\rho(v) \\
\mathcal{V}[\![\forall x : S. T]\!]_\rho(v) &\triangleq \exists e.\ (v =_\alpha \lambda x. e) \land \forall w.\ \triangleright \mathcal{V}[\![S]\!]_\rho(w) \Rightarrow \triangleright \mathcal{E}[\![T]\!]_{(\rho, x := w)}(e[x := w]) \\
\mathcal{V}[\![\{\mathsf{a} : T\}]\!]_\rho(v) &\triangleq \exists x, \bar{d}.\ (v =_\alpha \nu x. \{\bar{d}\}) \land \overline{\mathcal{D}}[\![\{\mathsf{a} : T\}]\!]_\rho(\bar{d}[x := v]) \\
\mathcal{V}[\![\{\mathsf{A} >: L <: U\}]\!]_\rho(v) &\triangleq \exists x, \bar{d}.\ (v =_\alpha \nu x. \{\bar{d}\}) \land \overline{\mathcal{D}}[\![\{\mathsf{A} >: L <: U\}]\!]_\rho(\bar{d}[x := v]) \\
\mathcal{V}[\![p.\mathsf{A}]\!]_\rho(v) &\triangleq \mathsf{wp_P}\ p[\rho]\ \{w.\ \exists \sigma, s, \psi.\ (w.\mathsf{A} \searrow (\sigma, s)) \land (s \rightsquigarrow_\sigma \psi) \land \triangleright \psi(v)\} \\
\mathcal{V}[\![p.\mathbf{type}]\!]_\rho(v) &\triangleq \mathsf{wp_P}\ p[\rho]\ \{w.\ v =_\alpha w\} \\
\mathcal{V}[\![\mu x. T]\!]_\rho(v) &\triangleq \mathcal{V}[\![T]\!]_{(\rho, x := v)}(v) \\
\mathcal{V}[\![\triangleright T]\!]_\rho(v) &\triangleq \triangleright \mathcal{V}[\![T]\!]_\rho(v)
\end{aligned}$$

Term interpretation $\boxed{\mathcal{E}[\![\_]\!]\_(\_) : \mathsf{Type} \to \mathsf{Env} \to \mathsf{Term} \to \mathsf{iProp}}$

$$\mathcal{E}[\![T]\!]_\rho(e) \triangleq \mathsf{wp}\ e\ \{\mathcal{V}[\![T]\!]_\rho\}$$

Environment interpretation $\boxed{\mathcal{G}[\![\_]\!](\_) : \mathsf{TyCtx} \to \mathsf{Env} \to \mathsf{iProp}}$

$$\begin{aligned}
\mathcal{G}[\![\varepsilon]\!](\rho) &\triangleq \mathsf{True} \\
\mathcal{G}[\![\Gamma, x : T]\!](\rho) &\triangleq \mathcal{G}[\![\Gamma]\!](\rho_{|\Gamma}) \land \mathcal{V}[\![T]\!]_\rho(\rho(x))
\end{aligned}$$

Semantic typing judgments

$$\begin{aligned}
\Gamma \vDash^i_{\mathbf{P}} p : T &\triangleq \forall \rho.\ \mathcal{G}[\![\Gamma]\!](\rho) \Rightarrow \triangleright^i \mathsf{wp_P}\ p[\rho]\ \{\mathcal{V}[\![T]\!]_\rho\} \\
\Gamma \vDash_{\mathbf{S}} e : T &\triangleq \forall \rho.\ \mathcal{G}[\![\Gamma]\!](\rho) \Rightarrow \mathcal{E}[\![T]\!]_\rho(e[\rho]) \\
\Gamma \mid x : V \vDash_{\mathbf{S}} \{\bar{d}\} : T &\triangleq \mathsf{wf}\ \bar{d} \land \forall \rho, \bar{d}_v.\ \mathsf{wf}\ \bar{d}_v \Rightarrow (\bar{d} \subseteq \bar{d}_v[x := \nu x. \{\bar{d}_v\}]) \Rightarrow \\
&\qquad \mathcal{G}[\![\Gamma, x : V]\!](\rho, x := \nu x. \{\bar{d}_v\}) \Rightarrow \overline{\mathcal{D}}[\![T]\!]_\rho(\bar{d}[\rho]) \\
\Gamma \vDash^i T_1 <: T_2 &\triangleq \forall \rho, v.\ \mathcal{G}[\![\Gamma]\!](\rho) \Rightarrow \triangleright^i \big(\mathcal{V}[\![T_1]\!]_\rho(v) \Rightarrow \mathcal{V}[\![T_2]\!]_\rho(v)\big) \\
\Gamma \vDash e : T &\triangleq \ \dot{\mid\!\Rrightarrow}(\exists e'.\ e \approx e' \land \Gamma \vDash_{\mathbf{S}} e' : T) \\
\Gamma \mid x : V \vDash \{\bar{d}\} : T &\triangleq \ \dot{\mid\!\Rrightarrow}(\exists \overline{d'}.\ \bar{d} \approx \overline{d'} \land \Gamma \mid x : V \vDash_{\mathbf{S}} \{\overline{d'}\} : T)
\end{aligned}$$

Fig. 9. The semantic model of gDOT. Relation wf $\bar{d}$ asserts that $\bar{d}$ contains no duplicate labels. Environment restriction $\rho_{|\Gamma}$ restricts ρ to entries in Γ.

Unlike term typing, path typing thus does not allow its subject to diverge (unlike in pDOT, paths in gDOT cannot loop; see Sec. 8 for further discussion).

5.3.2 Unstamped Semantic Typing Judgments. The unstamped semantic typing judgment $\Gamma \vDash e : T$ for terms is defined by lifting the stamped typing judgment $\Gamma \vDash_{\mathsf{s}} e : T$. It simply says that there exists a stamped term e that satisfies $\Gamma \vDash_{\mathsf{s}} e : T$ and equals e modulo type members. The update modality $\dot{\Rrightarrow}$ is needed to allocate saved predicates for type members.

Similarly, we define an unstamped semantic typing judgment $\Gamma \mid x : V \vDash \{\bar{d}\} : T$ for definition lists. We do not need to lift the judgments for path typing and subtyping, because (1) there is no distinction between unstamped and stamped types (2) while stamped paths exist, subjects of path typing do not contain values, so they do not differ in unstamped and stamped gDOT either.

5.3.3 Semantic Typing Lemmas and Fundamental Theorem. After having defined the semantic typing judgments, we can prove the *semantic typing lemmas*. Basically, for each typing rule, we replace $\vdash$ with $\vDash$. In fact, while designing gDOT, what we did was exactly the opposite — we first proved the semantic typing lemmas before turning gDOT into a syntactic type system.

The proofs of the bulk of the semantic type lemmas are fairly straightforward — the majority of the work was in devising the right interpretations of types. All proofs proceed by first proving a version of the typing lemma for the stamped semantic typing judgment, which we subsequently lift to the unstamped judgment; lifting is trivial except for (D-Typ-Abs). Typing rules with interesting proofs include (T-{}-I) and (Sel-<:). The proof of rule (T-{}-I) relies on (Löb) induction: to prove that the object $v \triangleq \nu x.\,\{\bar{d}\}$ satisfies $\mathcal{V}[\![\, \mu x.\, T \,]\!]_\rho(v)$, we can assume it satisfies $\triangleright(\mathcal{V}[\![\, \mu x.\, T \,]\!]_\rho(v))$. The proof of rule (Sel-<:) uses Iris's proof rule (Saved-Pred-Agree) for saved propositions. This proof also explains the $\triangleright$ in the rule (Sel-<:) — it appears in rule (Sel-<:) because it appears in (Saved-Pred-Agree). In general, the proofs of the semantic typing rules explain that types and function bodies only contain information later, hence introduction and elimination rules only require information under a later type former.

As expected for a semantic model based on logical relation, putting together the semantic typing lemmas, we prove the *fundamental theorem* (Theorem 5.3), *i.e.,* if $\Gamma \vdash e : T$, then $\Gamma \vDash e : T$.

Proof of Theorem 5.3. The theorem is proved by induction on term and definition typing judgments (after similar theorems on other judgments). Each case of the proof corresponds to a syntactic typing rule and follows from the corresponding semantic typing lemma. □

5.3.4 Adequacy. We outline the proof of adequacy (Theorem 5.4), *i.e.,* if $\vDash e : T$, then e is safe.

Proof of Theorem 5.4. We must show that any reduction $e \to_{\mathrm{t}}^{n} e_r$ produces a non-stuck term e_r. By definition of the unstamped judgment $\vDash e : T$, we obtain a stamped term e' with $\vDash_{\mathsf{s}} e' : T$ and $e \approx e'$. By definition of the stamped judgment and term interpretation, this gives $\mathsf{wp}\ e'\ \{\mathcal{V}[\![\, T \,]\!]_\rho\}$, so any reduction $e' \to_{\mathrm{t}}^{n} e'_r$ produces a non-stuck term e'_r, and by $e \approx e'$, term e_r is not stuck either.

The above reasoning is performed internally in Iris. Thus, we actually obtain non-stuckness of e_r under n laters, as laters are accumulated each time the definition of weakest preconditions is unfolded. Since non-stuckness is a meta theoretical (*i.e.,* Coq) proposition, we can eliminate the laters using Iris's soundness theorem and obtain non-stuckness of e_r at the meta-level. This proof resembles the adequacy proof of weakest preconditions in Iris [Krebbers et al. 2017a]. □

6 EXPRESSIVITY EVALUATION

We show that, despite gDOT's guardedness restrictions, we can encode both existing examples from the literature and new ones. All examples presented in this section, and additional ones, including all examples in Sec. 1-5 of the WadlerFest DOT paper [Amin et al. 2016], are mechanized in Coq.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{let } \mathit{bools} = \ldots \textbf{ in let } \mathit{lists} = \nu\, \mathit{lists}.\, \{} \\
\quad \mathsf{List} & >: \bot = \mu\, \mathit{list}. \\
 & \{\mathsf{A} >: \bot <: \top;\ \texttt{isEmpty} : \top \to \mathit{bools}.\mathsf{Bool};\ \texttt{head} : \top \to \mathit{list}.\mathsf{A};\ \texttt{tail} : \top \to \mathit{lists}.\mathsf{List} \wedge \{\mathsf{A} <: \mathit{list}.\mathsf{A}\}\} \\
\quad \texttt{nil} & : \triangleright\, \mathit{lists}.\mathsf{List} \wedge \{\mathsf{A} = \bot\} \\
 & = \nu\, \_.\, \{\mathsf{A} = \bot;\ \texttt{isEmpty} = \lambda\_.\ \mathit{bools}.\texttt{true};\ \texttt{head} = \lambda\_.\ \textbf{diverge};\ \texttt{tail} = \lambda\_.\ \textbf{diverge}\} \\
\quad \texttt{cons} & : \forall\, (x : \{\mathsf{S} <: \top\}).\, x.\mathsf{S} \to (\mathit{lists}.\mathsf{List} \wedge \{\mathsf{A} <: x.\mathsf{S}\}) \to \mathit{lists}.\mathsf{List} \wedge \{\mathsf{A} <: x.\mathsf{S}\} \\
 & = \lambda\, x\ \mathit{hd}\ \mathit{tl}.\ \nu\, \_.\, \{\mathsf{A} = x.\mathsf{S};\ \texttt{isEmpty} = \lambda\_.\ \mathit{bools}.\texttt{false};\ \texttt{head} = \lambda\_.\ \mathit{hd};\ \texttt{tail} = \lambda\_.\ \mathit{tl}\} \\
\multicolumn{2}{l}{\}\ \textbf{in}\ \ldots}
\end{array}
$$

Fig. 10. Covariant lists in gDOT using (elided for space) Church-encoded Booleans [Amin et al. 2016].

In Sec. 6.1 we describe the syntactic typing of an encoding of covariant lists, a highly recursive benchmark from the DOT literature. In Sec. 6.2 and Sec. 6.3 we show that our semantic model can be used beyond proving type soundness — we use it to demonstrate that gDOT enforces data abstraction, and apply that to our motivating example from the introduction (Sec. 1.1).

6.1 Covariant Lists

As it is standard in the DOT literature [Amin et al. 2016; Rapoport and Lhoták 2019; Rompf and Amin 2016], we encode the Scala type $\texttt{List}[T]$ of lists, together with its core methods. Our encoding, which is shown in Fig. 10, is mostly standard in DOT (except for the shaded parts, to which we return in a moment), but we summarize a few features of this encoding. Object *lists* defines an abstract type of lists List, together with constructors `nil` and `cons`. The type of lists defines a type member A representing the type of elements, together with accessor methods. The definition of *lists*.List is highly recursive: it uses self variables *lists* and *list* to refer to both itself and its own type member *list*.A. Since gDOT (like all DOT calculi) lacks exceptions, here and in later examples we let failing methods invoke an infinite loop **diverge** with type ⊥, like in other DOT papers [Amin et al. 2016; Rapoport and Lhoták 2019; Rompf and Amin 2016]. We encode $\texttt{List}[T]$ as $\mathit{lists}.\mathsf{List} \wedge \{\mathsf{A} <: T\}$. Similarly to lists in Scala, this encoding is *covariant*, *i.e.*, if T_1 $\texttt{<:}$ T_2 then $\texttt{List}[T1]$ $\texttt{<:}$ $\texttt{List}[T_2]$.

Type checking the body of `cons` relies on gDOT's rules (μ-<:) and (<:-μ) for subtyping of recursive types [Rompf and Amin 2016]. Since most other DOT variants [Amin et al. 2016; Rapoport et al. 2017; Rapoport and Lhoták 2019] do not support those rules, they require instead modifying the source code and inserting a spurious let-binding, to then use variants of (P-μ-I) and (P-μ-E).

The only unusual guardedness restriction of gDOT is the use of a later in front of the type of `nil`. Since `nil` is defined as a value member, we cannot use coercions; and since the type of self variable *lists* is guarded during construction, we cannot derive bounds for *lists*.List but only for $\triangleright$ *lists*.List. This restriction could be avoided by thunking `nil` (*i.e.*, making it a method). In the present encoding, the later can be removed at the client side via a **coerce**.

6.2 Positive Numbers

In this and next section, we demonstrate gDOT's support for data abstraction through two examples that use escape hatches, such as unsafe casts. These examples' safety depends on gDOT's ability to maintain class invariants. While such examples cannot be typed syntactically, we show they satisfy our model's semantic typing judgment, by dropping down to the definition of semantic typing in Iris, similarly to the RustBelt model of Rust [Jung et al. 2018a, 2020]. Since semantic typing judgments can be combined with any syntactically well-typed code, this means that the examples can be used safely in any well-typed context.

$$\begin{array}{ll}
\mathit{posSemT} \in \mathsf{SemType} & \triangleq \lambda\, \rho, v.\ \exists\, n : \mathbb{Z}.\ v = \underline{n} \land n > 0 \\
\quad \textbf{let}\ \mathit{positives} & = \nu\, \mathit{positives}.\ \{ \\
\qquad \mathsf{Pos} & >: \bot <: \mathtt{Int} = \mathit{posSemT} \\
\qquad \mathtt{mkPos} & : \mathtt{Int} \to \mathit{positives}.\mathsf{Pos} = \lambda\, m.\ \ \textbf{if}\ m > 0\ \textbf{then}\ m\ \textbf{else diverge} \\
\qquad \mathtt{div} & : \mathtt{Int} \to \mathit{positives}.\mathsf{Pos} \to \mathtt{Int} = \lambda\, m\ n.\ \ m / (\,\textbf{coerce}\ n) \\
\quad \}\ \textbf{in} \ldots &
\end{array}$$

Fig. 11. A module for positive numbers and safe division using abstract types.

As a warm-up, and to demonstrate the use of semantic types, we show that object *positives* in Fig. 11 is semantically well-typed. This object defines a type member Pos of positive numbers, and methods to create (`mkPos`) and consume them (`div`). The method `mkPos` represents a "smart constructor": it returns the input number *m* if positive, and fails by looping otherwise. The method `div` uses an *unsafe* division operator, which gets stuck when applied to the divisor 0.[7] This method (hence, object *positives*) cannot be typed in the syntactic type system of (g)DOT, because its safety relies on functional correctness (*i.e.,* the argument being non-zero).

Yet, we can prove that *positives* is *semantically typed*. While the class invariant of Pos cannot be expressed through a syntactic type, we can express it as the logical predicate *posSemT*. To prove semantic typing of *positives*, we unfold the semantic typing judgment of our gDOT model and perform a manual proof in Iris. Let us walk through that proof. First, we need to prove that Pos respects its type bounds ⊥ and `Int`. This holds trivially because positive integers are integers. Next, we should prove semantic typing of `mkPos`. For that, we need to show that if the conditional succeeds, the argument satisfies *posSemT*. Finally, we should prove semantic typing of `div`. Since its argument *n* has abstract type *positives*.Pos, it satisfies semantic type *posSemT*. Note that since type members are modeled using saved predicates in Iris, method `div` uses a coercion, allowing us to strip a later when acquiring the semantic type *posSemT* of type member *positives*.Pos.

Thanks to the Iris framework, we do not have to deal with explicit step-indexing. All proofs are carried out using Iris's support for abstract step-indexing. Moreover, to streamline semantic typing proofs such as the above, we have generalized semantic typing judgments to semantic types in our Coq mechanization, which enable us to reuse our typing lemmas both here and in Sec. 6.3.

6.3 Mutual Information Hiding

We now return to our motivating example from the introduction (Sec. 1.1). As discussed in Sec. 4, we have shown in Coq that the gDOT version (Fig. 2) of the Scala code (Fig. 1) is syntactically well-typed. However, method `typeFromTypeRefUnsafe` (which is present in the Scala version, but not in the DOT version), cannot be shown to be syntactically well-typed in any DOT calculus — it uses an unsafe cast whose safety crucially relies on (g)DOT's support for data abstraction. Using gDOT's semantic model we show that this example is in fact semantically well-typed. This demonstrates the flexibility of semantic typing, and shows that gDOT enforces the data abstraction that mutual information hiding should provide.

Method `typeFromTypeRefUnsafe` in Fig. 1 retrieves an actual `types.Type` by invoking the `get` method on `t.symb.tpe`, which has type `Option[types.Type]`. In Scala, invoking `Option`[T]'s method `get` on `None` will trigger an exception. However, since gDOT lacks exceptions, and to show gDOT's support for data abstraction, we model (unlike the Scala standard library) `get` as a function from `Some`[T] to T, where `Some`[T] is a subtype of `Option`[T] that has a `get` function. Hence, to call `get`,

[7] Whereas we used Church encoded Booleans in Sec. 6.1, the version of gDOT that we mechanized in Coq in fact has primitive support for Booleans and integers, which we use in this section.

```
assert c ≜ if c then 0 else diverge;
None ≜ {isEmpty : true.type; . . .}
Some ≜ μ some. {
  A        >: ⊥ <: ⊤
  isEmpty  : false.type   // The only value in singleton type false.type is false.
  pmatch   : ∀ (x : {U <: ⊤}). x.U → (some.A → x.U) → x.U
  get      : ▶ some.A
}
let options : {Option <: None ∨ Some; . . .} = . . . in
let pcore = ν pcore. {
  types = ν types. {
    Type             >: ⊥ = ⊤
    TypeTop          >: ⊥ = types.Type
    newTypeTop       : ⊤ → types.TypeTop = λ_. ν_. {}
    TypeRef          >: ⊥ <: types.Type ∧ {symb : pcore.symbols.Symbol}
                     = types.Type ∧ {symb : (pcore.symbols.Symbol ∧ {tpe : Some})}
    newTypeRef       : pcore.symbols.Symbol → types.TypeRef
                     = λs. { assert(¬( coerce s).tpe.isEmpty); ν_. {symb = s} }
    typeFromTypeRef  : types.TypeRef → types.Type =
                     = λt. { coerce ( coerce ( coerce ( coerce t).symb).tpe.get) }
  } // symbols is unchanged
} in . . .
```

Fig. 12. The (simplified) fragment of Dotty from Fig. 1 in gDOT (**assert**, **None**, and **Some** are abbreviations).

we first must unsafely cast `tpe` via `tpe.asInstanceOf[Some[types.Type]]` to get a value of type `Some[types.Type]`. Although this cast cannot be typed syntactically, it is safe due to the `assert` in constructor `TypeRef`. In turn, safety of this `assert` relies on `Option`'s class invariant: `isEmpty` only returns `false` on instances of the `Some` constructor.

We encode the Scala example from Fig. 1 in gDOT as shown in Fig. 12. As usual in gDOT, we use coercions when unfolding abstract types to ensure guardedness. More importantly, we express the class invariants of types *options*.Option and pcore.types.TypeRef by defining them to stricter types than in Scala. In particular, the upper bound of Option formalizes the informal invariants of *options*'s public API using union and singleton types. An instance of Option is then either an instance of **None**, exposing an isEmpty method that returns **true**, or an instance of **Some**, exposing an isEmpty method that returns **false** and a get method that returns the contained value.

Thanks to gDOT's support for mutual information hiding, one can also expose class invariants locally, and hide them by using subsumption. This is used for TypeRef, whose class invariant guarantees that symb.tpe has type **Some** containing method get, but this is hidden outside types. The more precise definition of TypeRef makes typeFromTypeRef syntactically well-typed (hence the change of name), but makes newTypeRef syntactically ill-typed, so we prove it well-typed *semantically*. In this proof, we take the result v of (**coerce** s).tpe, show it has type **None** ∨ **Some**, and reason by cases on this union type. If v has type **None**, newTypeRef diverges and is thus safe. If v has type **Some**, the return value of newTypeRef will have the correct type TypeRef. This concludes our informal proof sketch, which we have mechanized in Coq. Parts of the typing derivation are constructed syntactically; in those parts, we needed various distributivity rules, including rule (Distr-∧-∨-<:) (see Sec. 4.4) to distribute intersections over union and show (**None** ∨ **Some**) ∧ {A >: ⊥ <: *pcore*.types.Type} <: **None** ∨ (**Some** ∧ {A >: ⊥ <: *pcore*.types.Type}).

7 COQ MECHANIZATION

We mechanized gDOT and its semantic soundness proof in Coq, using the Iris framework. The mechanization helped us gain confidence in our ideas, and to evolve our definitions and proofs. Our mechanization crucially relies on the MoSeL tactic language [Krebbers et al. 2018, 2017b], which provides tactics for reasoning at the level of abstract step-indexing with Iris's modalities.

We mechanized binding through de Bruijn indexes and parallel substitution, using the Autosubst 1 library [Schäfer et al. 2015]. While DOT binding does not fit perfectly with Autosubst 1 (unlike Autosubst 2 [Stark et al. 2019]), we were able to use Autosubst 1 by defining substitution by hand, while reusing Autosubst 1's tactics for deciding binding lemmas. Due to Autosubst 1's limitations, path substitution is defined separately. The only axiom we use is functional extensionality (needed by Autosubst).

Overall, our gDOT mechanization currently consists of 14.774 lines of Coq code, of which 5.610 lines are for examples and support code (including derived typing rules) and 9.164 for the actual soundness proof. It includes some language-generic components (2.761 lines). The mechanization of the gDOT semantic model (5.555 lines) consists of the definition of the gDOT language, operational semantics and bisimulation (2.562 lines), and the logical relation, semantic typing lemmas and adequacy theorem (2.993 lines). The mechanization of the gDOT syntactic type system defines the syntactic type system, some derived rules, and proves the fundamental theorem (848 lines).

8 RELATED WORK

pDOT. The variant of the DOT calculus that is closest to gDOT is pDOT, introduced by Rapoport and Lhoták [2019]. The rule (D-Val-New) of gDOT is an "alternative design" they considered for pDOT [Rapoport and Lhoták 2019, Sec. 4.2.2]. Unlike Scala, pDOT considers paths as normal forms. Instead, gDOT lets paths reduce, but ensures they have a normal form.

Normalization for $D_{<:}$. We were inspired by Wang and Rompf [2017], who use logical relations to prove normalization of a DOT subset including $D_{<:}$. They prove normalization (in a way that implies type safety), which they argue is important for paths. However, for proving type safety of Scala (which in itself is not normalizing), it is sufficient to prove normalization of paths only. Indeed, our path typing judgment implies normalization through the use of total weakest preconditions in Iris (see Sec. 5.2.3). Moreover, their model imposes guardedness restrictions on μ-types instead of abstract types. Those guardedness restrictions are more severe than gDOT's — they only allow for a weak elimination rule for μ-types, reminiscent of System F-style weak existentials. While their results also imply type safety for the language they study, it is unclear how to adapt their technique to prove type safety of a Turing-complete (*i.e.,* non-normalizing) variant of the language.

Coinductive Type Systems. Brandt and Henglein [1998] define subtyping for recursive types using a coinductive formulation of subtyping, which resembles our typing rule for object creation, and our use of Löb induction. That is, to prove a judgment J (such as type equality or subtyping), they allow using J as an assumption, but forbid using J immediately. One might suspect that a coinductive formulation of DOT, and of rule (T-{}-I) in particular, might allow making (D-Typ-Abs) sound without using later. However, DOT's μ-types (and Amin et al.'s refinements [2012]) differ from standard recursive types, and resemble more closely recursively defined signatures [Crary et al. 1999], Cedille's ι-types [Fu and Stump 2014], and dependent intersections [Kopylov 2003].

Logical relations for Predicative Type Members. Logical relation models are available for other type systems with features similar to Scala type members, such as ML modules [Crary 2017] and type theory. However, such type systems avoid the challenges we face because they feature a universe hierarchy and predicative/stratified type members/Σ-types: if a value v contain a type in a certain universe, the type of v lives in a larger universe [Harper and Mitchell 1993], eschewing the need for stratification via step-indexing.

Logical Relations in Iris. Logical relations have been studied extensively in the context of Iris — for type soundness [Jung et al. 2018a; Krebbers et al. 2017b], program refinements [Frumin et al. 2018; Krebbers et al. 2017b; Krogh-Jespersen et al. 2017; Tassarotti et al. 2017; Timany et al. 2018], robust safety [Swasey et al. 2017], and non-interference [Frumin et al. 2020]. This paper studied a number of novel features that are have not been studied in the Iris context before: dependent types, impredicative type members, union and intersection types, and the combination of non-termination (for terms) and termination (for paths). To support these features, we developed novel techniques, such as stamping, and combine weakest preconditions for partial and total correctness.

Virtual Classes and Impredicative Type Members. Type members in DOT and Scala eschew the sort of universe hierarchy described in the previous paragraph: we say they feature *impredicative* type members. Impredicative type members also feature in other type systems with path-dependent types or virtual classes [Clarke et al. 2007; Ernst et al. 2006].

9 FUTURE WORK

Annotation inference and type checking. DOT calculi are not meant to be programmed in directly, but should be considered as an elaboration target for type-preserving translation from subsets of Scala. Type checking of DOT is conjectured to be undecidable, like $D_{<:}$ [Hu and Lhoták 2020]. gDOT additionally requires inserting later ($\triangleright$) and coercion (**coerce**) annotations. Future work could investigate inference of these annotations, either directly [Severi 2019], or by translating from a Scala subset with decidable type checking [Cremet et al. 2006] into gDOT or a suitable variant.

Expressivity. The programs we prove safe are decorated by no-op coercions (**coerce**). We conjecture that removing these coercions preserves safety, but we leave a proof for future work.

Amin et al. [2016] prove that all $F_{<:}$ programs can be translated into DOT. Due to the presence of the $\triangleright$ operator and the **coerce** annotations, it is unclear how to create a translation from either (p)DOT or $F_{<:}$ into gDOT. However, we have been able to translate many given $F_{<:}$ and DOT examples into gDOT by hand by adding a sufficient number of $\triangleright$ and **coerce** annotations. We thus conjecture that there exists a whole-program encoding of $F_{<:}$ programs into gDOT.

Additional features. We are investigating support for higher-kinded types, by modeling type arguments as values. The latest work in this direction [Stucki 2016, 2017] ran into strong challenges and a counterexample to soundness (luckily, not affecting Scala). We conjecture that our techniques scale directly to this form of higher kinds, and that gDOT's existing guardedness restrictions already rule out this counterexample.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback, Sandro Stucki, Tiark Rompf, Nada Amin, Marianna Rapoport, Samuel Grütter, Ondřej Lhoták, Andreas Rossberg and Dimitrios Vytiniotis for helpful discussions on DOT, and Derek Dreyer for first directing the first author to Iris.

During this project Amin Timany was a postdoctoral fellow of the Flemish research fund (FWO) and worked at the imec-DistriNet research group at KU Leuven, Belgium. Robbert Krebbers was supported by the Dutch Research Council (NWO), project 016.Veni.192.259. This work was also supported by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation and by the ModuRes Sapere Aude Advanced Grant from The Danish Council for Independent Research for the Natural Sciences (FNU).

REFERENCES

Amal Ahmed. 2004. *Semantics of Types for Mutable State.* Ph.D. Dissertation. Princeton University.

Pierre America and Jan J. M. M. Rutten. 1989. Solving reflexive domain equations in a category of complete metric spaces. *JCSS* 39, 3 (1989), 343–375.

Nada Amin. 2016. *Dependent Object Types.* Ph.D. Dissertation. EPFL.

Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The essence of dependent object types. In *WadlerFest (LNCS)*, Vol. 9600. 249–272.

Nada Amin, Adriaan Moors, and Martin Odersky. 2012. Dependent object types. In *FOOL*.

Nada Amin and Tiark Rompf. 2017. Type soundness proofs with definitional interpreters. In *POPL*. 666–679.

Andrew W. Appel and David A. McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS* 23, 5 (2001), 657–683.

Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. 2007. A very modal model of a modern, major, general type system. In *POPL*. 109–122.

Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. 2011. Step-indexed Kripke models over recursive worlds. In *POPL*. 119–132.

Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. 2010. The category-theoretic solution of recursive metric-space equations. *TCS* 411, 47 (2010), 4102–4122.

Michael Brandt and Fritz Henglein. 1998. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae* 33, 4 (1998), 309–338.

Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. 2007. Tribe: a simple virtual class calculus. In *AOSD*, Vol. 208. 121–134.

Karl Crary. 2017. Modules, abstraction, and parametric polymorphism. In *POPL*. 100–113.

Karl Crary, Robert Harper, and Sidd Puri. 1999. What is a recursive module?. In *PLDI*. 50–63.

Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. 2006. A core calculus for Scala type checking. In *MFCS (LNCS)*, Vol. 4162. 1–23.

Erik Ernst, Klaus Ostermann, and William R. Cook. 2006. A virtual class calculus. In *POPL*. 270–282.

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency. In *LICS*. 442–451.

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2020. Compositional Non-Interference for Fine-Grained Concurrent Programs. To appear in S&P'21.

Peng Fu and Aaron Stump. 2014. Self types for dependently typed lambda encodings. In *RTA-TLCA (LNCS)*, Vol. 8560. 224–239.

Paolo G. Giarrusso. 2019. Can we prove that type constructors are "distributive"? Github issue, https://web.archive.org/web/20200304175526/https://github.com/lampepfl/dotty-feature-requests/issues/51, archived on 04 March 2020.

Paolo G. Giarrusso, Léo Stefanesco, Amin Timany, Lars Birkedal, and Robbert Krebbers. 2020. Scala Step-by-Step: Soundness for DOT with Step-Indexed Logical Relations in Iris – Extended Version and Coq Mechanization. Available online at https://dot-iris.github.io/, archived version of the Coq development available at https://doi.org/10.5281/zenodo.3926703.

Robert Harper and Mark Lillibridge. 1994. A type-theoretic approach to higher-order modules with sharing. In *POPL*. 123–137.

Robert Harper and John C. Mitchell. 1993. On the type structure of Standard ML. *TOPLAS* 15, 2 (1993), 211–252.

Jason Z. S. Hu and Ondřej Lhoták. 2020. Undecidability of $D_{<:}$ and its decidable fragments. *PACMPL* 4, POPL (2020), 9:1–9:30.

DeLesley S. Hutchins. 2010. Pure subtype systems. In *POPL*. 287–298.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. RustBelt: Securing the foundations of the Rust programming language. *PACMPL* 2, POPL (2018), 66:1–66:34.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2020. Safe systems programming in Rust: The promise and the challenge. To appear in CACM.

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *ICFP*. 256–269.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018b. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *JFP* 28 (2018), e20.

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*. 637–650.

Ifaz Kabir and Ondřej Lhoták. 2018. κDOT: Scaling DOT with mutation and constructors. In *SCALA@ICFP*. 40–50.

Alexei Kopylov. 2003. Dependent intersection: A new way of defining records in type theory. In *LICS*. 86–95.

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A general, extensible modal framework for interactive proofs in separation logic. *PACMPL* 2, ICFP (2018), 77:1–77:30.

Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017a. The essence of higher-order concurrent separation logic. In *ESOP (LNCS)*, Vol. 10201. 696–723.

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017b. Interactive proofs in higher-order concurrent separation logic. In *POPL*. 205–217.

Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. 2017. A relational model of types-and-effects in higher-order concurrent separation logic. In *POPL*. 218–231.

Robin Milner. 1978. A theory of type polymorphism in programming. *JCSS* 17, 3 (1978), 348–375.

Hiroshi Nakano. 2000. A Modality for Recursion. In *LICS*. 255–266.

Abel Nieto. 2017. Towards algorithmic typing for DOT (short paper). In *SCALA@SPLASH*. 2–7.

Martin Odersky. 2016. DOT with higher-kinded types. Github discussion, https://web.archive.org/web/20200304175613/https://gist.github.com/odersky/36aee4b7fe6716d1016ed37051caae95, archived on 04 March 2020.

Martin Odersky, Guillaume Martres, and Dmitry Petrashko. 2016. Implementing higher-kinded types in Dotty. In *SCALA@SPLASH*. 51–60.

Marianna Rapoport, Ifaz Kabir, Paul He, and Ondřej Lhoták. 2017. A simple soundness proof for dependent object types. *PACMPL* 1, OOPSLA (2017), 46:1–46:27.

Marianna Rapoport and Ondřej Lhoták. 2016. *Mutable WadlerFest DOT*. Technical Report. University of Waterloo. http://arxiv.org/abs/1611.07610

Marianna Rapoport and Ondřej Lhoták. 2019. A path to DOT: formalizing fully path-dependent types. *PACMPL* 3, OOPSLA (2019), 145:1–145:29.

Tiark Rompf and Nada Amin. 2016. Type soundness for dependent object types (DOT). In *OOPSLA*. 624–641.

Steven Schäfer, Tobias Tebbi, and Gert Smolka. 2015. Autosubst: reasoning with de Bruijn terms and parallel substitutions. In *ITP (LNCS)*, Vol. 9236. 359–374.

Paula Severi. 2019. A light modality for recursion. *LMCS* 15, 1 (2019).

Kathrin Stark, Steven Schäfer, and Jonas Kaiser. 2019. Autosubst 2: reasoning with multi-sorted de Bruijn terms and vector substitutions. In *CPP*. 166–180.

Sandro Stucki. 2016. DOT with higher-kinded types — A sketch. Github discussion, https://web.archive.org/web/20200304175148/https://gist.github.com/sstucki/3fa46d2c4ce6f54dc61c3d33fc898098, archived on 04 March 2020.

Sandro Stucki. 2017. *Higher-Order Subtyping with Type Intervals*. Ph.D. Dissertation. School of Computer and Communication Sciences, École polytechnique fédérale de Lausanne, Lausanne, Switzerland. https://doi.org/10.5075/epfl-thesis-8014 EPFL thesis no. 8014.

David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and compositional verification of object capability patterns. *PACMPL* 1, OOPSLA (2017), 89:1–89:26.

Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A Higher-Order Logic for Concurrent Termination-Preserving Refinement. In *ESOP (LNCS)*, Vol. 10201. 909–936.

Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. 2018. A logical relation for monadic encapsulation of state: Proving contextual equivalences in the presence of runST. *PACMPL* 2, POPL (2018), 64:1–64:28.

Fei Wang and Tiark Rompf. 2017. Towards strong normalization for dependent object types (DOT). In *ECOOP (LIPIcs)*, Vol. 74. 27:1–27:25.

Yanpeng Yang and Bruno C. d. S. Oliveira. 2017. Unifying typing and subtyping. *PACMPL* 1, OOPSLA (2017), 47:1–47:26.

APPENDIX

A OMITTED TYPING RULES

Fig. 13 and 14 display the typing rules that are omitted in Fig. 6 and 7. These rules are described either only in text or not at all. The subtyping rules in Fig. 13 allow distributing intersections over other types, and unions over value members. The reverse direction of these subtyping rules is derivable. Additionally, Fig. 15 fully defines the type equality judgment $T_1 \equiv T_2$, whereas Fig. 7 only includes (without names) rules ($\equiv$-$\triangleright$-$\land$, $\equiv$-$\triangleright$-$\lor$, $\equiv$-$\triangleright$-μ), and omits all other type equality rules.

$$\frac{}{\Gamma \vdash^i (\forall x : S.\, T_1) \land (\forall x : S.\, T_2) <: \forall x : S.\, T_1 \land T_2}\;(\textsc{Distr-}\land\textsc{-}\forall_1\textsc{-<:})$$

$$\frac{}{\Gamma \vdash^i (\forall x : S_1.\, T) \land (\forall x : S_2.\, T) <: \forall x : (S_1 \lor S_2).\, T}\;(\textsc{Distr-}\land\textsc{-}\forall_2\textsc{-<:})$$

$$\frac{}{\Gamma \vdash^i \{\mathsf{a} : T_1\} \land \{\mathsf{a} : T_2\} <: \{\mathsf{a} : T_1 \land T_2\}}\;(\textsc{Distr-}\land\textsc{-Fld-<:})$$

$$\frac{}{\Gamma \vdash^i \{\mathsf{A} >: L_1 <: U_1\} \land \{\mathsf{A} >: L_2 <: U_2\} <: \{\mathsf{A} >: (L_1 \lor L_2) <: (U_1 \land U_2)\}}\;(\textsc{Distr-}\land\textsc{-Typ-<:})$$

$$\frac{}{\Gamma \vdash^i \{\mathsf{a} : T_1 \lor T_2\} <: \{\mathsf{a} : T_1\} \lor \{\mathsf{a} : T_2\}}\;(\textsc{Distr-}\lor\textsc{-Fld-<:})$$

Fig. 13. Additional subtyping rules for distributivity, which are new in gDOT, and not present in pDOT.

$$\frac{}{\triangleright T \gg_{\triangleright} T}\;(\textsc{Strip-Later}) \qquad \frac{}{T \gg_{\triangleright} T} \qquad \frac{T_1 \gg_{\triangleright} U_1 \qquad T_2 \gg_{\triangleright} U_2}{T_1 \land T_2 \gg_{\triangleright} U_1 \land U_2} \qquad \frac{T_1 \gg_{\triangleright} U_1 \qquad T_2 \gg_{\triangleright} U_2}{T_1 \lor T_2 \gg_{\triangleright} U_1 \lor U_2}$$

$$\frac{T_1 \gg_{\triangleright} T_2}{\triangleright T_1 \gg_{\triangleright} \triangleright T_2} \qquad \varepsilon \gg_{\triangleright} \varepsilon \qquad \frac{T_1 \gg_{\triangleright} T_2 \qquad \Gamma_1 \gg_{\triangleright} \Gamma_2}{\Gamma_1, x : T_1 \gg_{\triangleright} \Gamma_2, x : T_2}$$

Fig. 14. Rules for later stripping judgments $T_1 \gg_{\triangleright} T_2$ and $\Gamma_1 \gg_{\triangleright} \Gamma_2$, used in (T-$\forall$-I-Strong).

$$\triangleright (T_1 \land T_2) \equiv \triangleright T_1 \land \triangleright T_2\;(\equiv\text{-}\triangleright\text{-}\land) \qquad \triangleright (T_1 \lor T_2) \equiv \triangleright T_1 \lor \triangleright T_2\;(\equiv\text{-}\triangleright\text{-}\lor) \qquad \triangleright (\mu x.\, T) \equiv \mu x.\, \triangleright T\;(\equiv\text{-}\triangleright\text{-}\mu)$$

$$\frac{T_1 \equiv T_2}{T_2 \equiv T_1}\;(\equiv\textsc{-Sym}) \qquad \frac{T_1 \equiv T_2 \qquad T_2 \equiv T_3}{T_1 \equiv T_3}\;(\equiv\textsc{-Trans}) \qquad \top \equiv \top \qquad \bot \equiv \bot \qquad \frac{T_1 \equiv T_2 \qquad U_1 \equiv U_2}{T_1 \land U_1 \equiv T_2 \land U_2}$$

$$\frac{T_1 \equiv T_2 \qquad U_1 \equiv U_2}{T_1 \lor U_1 \equiv T_2 \lor U_2} \qquad \frac{S_1 \equiv S_2 \qquad T_1 \equiv T_2}{\forall x : S_1.\, T_1 \equiv \forall x : S_2.\, T_2} \qquad \frac{T_1 \equiv T_2}{\{\mathsf{a} : T_1\} \equiv \{\mathsf{a} : T_2\}}$$

$$\frac{L_1 \equiv L_2 \qquad U_1 \equiv U_2}{\{\mathsf{A} >: L_1 <: U_1\} \equiv \{\mathsf{A} >: L_2 <: U_2\}} \qquad p.\mathsf{A} \equiv p.\mathsf{A} \qquad p.\mathbf{type} \equiv p.\mathbf{type} \qquad \frac{T_1 \equiv T_2}{\mu x.\, T_1 \equiv \mu x.\, T_2} \qquad \frac{T_1 \equiv T_2}{\triangleright T_1 \equiv \triangleright T_2}$$

Fig. 15. Rules for type equality judgment $T_1 \equiv T_2$, used in (<:-Eq). Unnamed rules are congruence rules.